SECTOR BASED CLUSTERING & ROUTING

By

SUDHEER KRISHNA CHIMBLI VENKATA

Bachelor of Technology

Jawaharlal Nehru Technological University

Hyderabad, India

2002

Submitted to the faculty of the
Graduate college of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
December, 2004

SECTOR BASED CLUSTERING & ROUTING

Thesis Approved:

Dr. JOHNSON P THOMAS

Thesis Adviser

Dr. G. E. HEDRICK

Dr. ISTVAN JONYER

Dr. GORDON EMSILE

Dean of the Graduate College

PREFACE

The role of communications is prominent in current management strategies. This has led the research community to shift concentration from effective communication in infrastructure networks to infrastructure-less networks. A mobile ad hoc wireless network is one such infrastructure-less network that aide in communication. As all the nodes are mobile in this network, security and power are the issues associated with them. They are many routing protocols that try to address either one of these issues, but do not consider the efficiency contributed by clustering algorithms. In sector based clustering, the concept of sectors is introduced based on which clustering in mobile ad hoc wireless networks is done. Along with the advantages of clustering, sector based clustering provides an architecture using, which any stochastic routing algorithm like sector based routing will be able to route information securely and conserve power of nodes. The power drain occurring in most frequently used routes in certain routing protocols can be eliminated and thus the network survivability is increased.

ACKNOWLEDGEMENTS

I am in debt for my adviser Dr. Johnson P Thomas for recognizing my potential and accepting me as his assistant. His guidance and unremitting encouragement throughout my thesis work were very supportive. The presented work is an outcome of his support, motivation, and precious time.

My thanks to Dr. G. E. Hedrick and Dr. Istvan Jonyer for serving on my graduate committee and supporting me. Their comments and suggestions are greatly appreciated.

My heartfelt thanks go to my family members in INDIA for their extreme love and continuing support for my education. I love to thank my roommates and friends for their moral support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

Communication plays a crucial role in the current decision making strategies. In the last decade research in communications has allowed people to communicate with each other from anywhere in the world. Cellular communications provided by different service providers using technologies like Global System for Mobile communication (GSM), Code Division Multiple Access (CDMA), and Time Division Multiple Access (TDMA) is the outcome of the last two decades of research. Although these mobile communication technologies are wireless, they require fixed infrastructures to support their operations. In many applications and scenarios (battlefield for example) it is not always possible to have a fixed infrastructure and in such environments, the communications system must work in the absence of any infrastructure. Mobile ad hoc wireless network is an example of infrastructureless communication networks [Royer and Toh 99].

Mobile Ad hoc Networks (MANET) arose from the Defense Advanced Research Projects Agency (DARPA) research on packet radio networks [Wireless 03]. MANETs have a wide range of applications that range from the battlefield to disaster zones. There are many customized algorithms for MANET that were developed from algorithms used in fixed wired networks. The mobility factor associated with MANETs make these algorithms inappropriate and inefficient. The Power associated with a MANET node is a crucial feature to be considered during development of any protocol for MANETs, as the nodes have limited battery power supply. Power conservation is essential as it ensure the survivability of a network. The algorithms should ensure that a node does not die out early and the network stays connected (or survives) for a longer period of time. As MANETs have a big role in military applications, security is the other feature that researchers have concentrated on.

1.1 Problem Statement

As MANETs have no fixed infrastructure, all messages have to be routed through the nodes in the network. Many clustering and routing algorithms have been developed for MANETs. Moreover, most of the existing routing algorithms do not utilize the efficiency that can be obtained by clustering a network. Clustering process involves in grouping network nodes and helps in reducing the overhead messages that help in establishing routes. Furthermore, there is a trade-off between providing security and conserving the power of a node. In current approaches, clustering and routing algorithms are designed specifically for either providing security or conserving power. It is very difficult to improve both security and minimize power consumption, as typically one is achieved at the expense of the other. In this thesis, we propose a novel Sector Based Clustering approach to provide additional security to a node during routing of information as well as increase the overall network lifetime by keeping the nodes alive for longer periods. Sectors regulate the number of members present in each cluster and also permit to have nodes as members of multiple clusters in a controlled way. This ensures that power is conserved. The Sector Based Routing algorithm uses Sector Based

Clustering approach and provides a more efficient way of routing information. The Sector Based Routing algorithm enhances the security of data that is transmitted between the source and destination as well as increase the survivability of the network.

1.2 Thesis Outline

In chapter 2, a brief introduction to ad hoc networks is given along with their applications and problems. The different basic clustering and routing algorithms that have been developed for MANETs are given in chapters 3 and 4 respectively. Our proposed approach for sector based clustering and routing is given in chapter 5. The implementation is described in chapter 6 and, simulation results and observations presented in chapter 7. Chapter 8 concludes the thesis and provides suggestions for future research.

CHAPTER II

MOBILE AD HOC NETWORKS

This chapter gives a brief introduction about mobile ad hoc networks along with their applications. It also highlights some of the problems associated with them.

2.1 Introduction

Ad hoc Networks were previously known as packet radio networks. They are also known as infrastructureless network, as there is no fixed network dedicated for them. All the nodes are considered to be mobile that communicate through wireless signals. The nodes within each other's radio range communicate directly via wireless links, while those that are far apart rely on other nodes to relay messages as routers. There is no fixed topology for ad hoc networks; they are subjected to lot of changes because of the mobility factor. This is shown in figure 2.1 below.

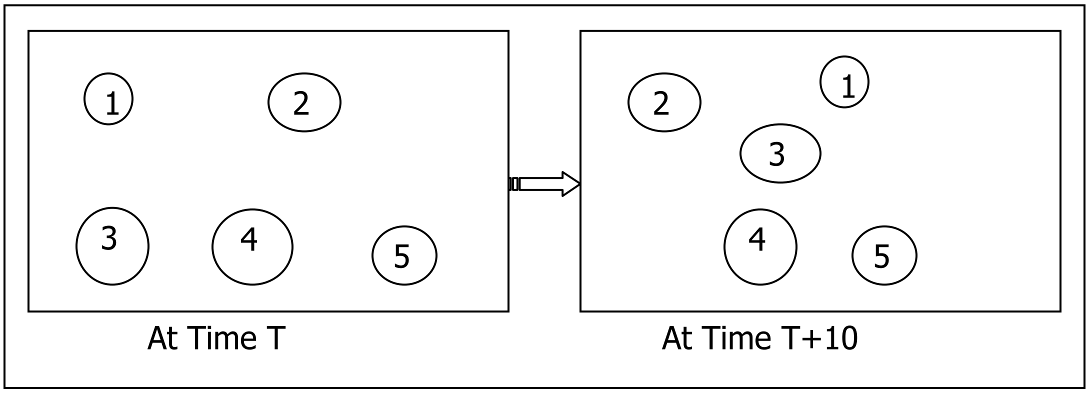

Figure 2.1. Ad hoc Network variable topology.

A collection of communication devices or nodes that wish to communicate, but do not have a fixed infrastructure and information about available links form an ad hoc network. The individual nodes determine which other nodes they are able to communicate with. The communication range for each node depends on its radio signal strength, so it may not be possible for a node to communicate directly with every other node present in the network [Deng et al. 02]. The communication between nodes may be done using multi hop packet relays as show in figure 2.2. Clustering and Routing play an important role in MANETs.

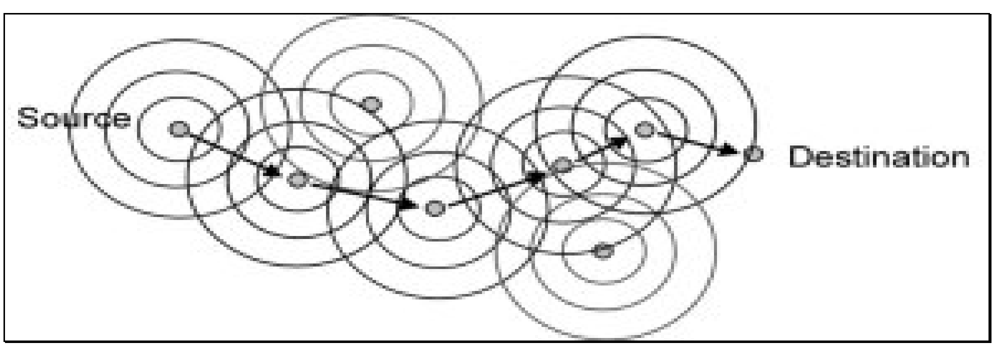

Figure 2.2. Ad Hoc Network–Relaying traffic between Source and Destination [from Wireless 03].

2.2 Applications

MANET do not require any initial infrastructure setup, so there is no initial setup delay in order to make the network operational. In order to make a group of autonomous nodes take part in networking they need to exchange initial data among them, and this takes very minimal amount of time. The above feature of a MANET makes it suitable in situations where there is no infrastructure available, cannot be trusted, or cannot be relied on in times of emergency. These include battlefield situations, military applications, and other emergency and disaster circumstances. Figure 2.3 shows how a MANET is applicable to these situations.

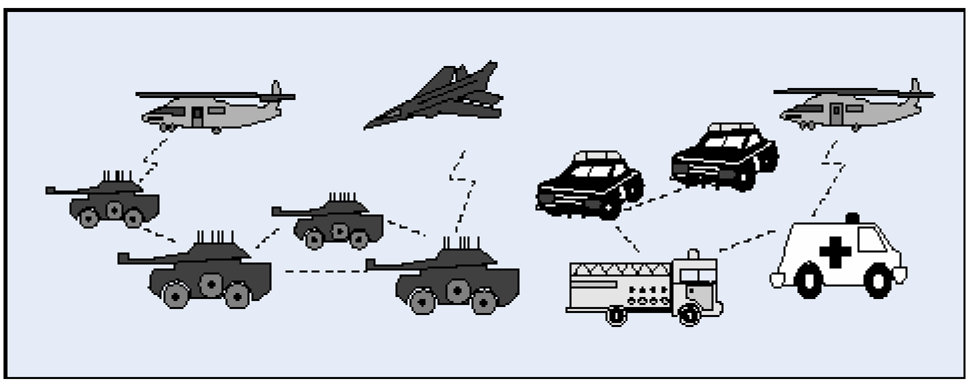

Figure 2.3. Example applications of MANET [from Deng et al. 02].

Ad hoc Networks are developed to provide instant connectivity across different communication devices irrespective of their location and underlying technology. This is considered to be an important goal in networking and is shown for a military situation in figure 2.4 below.

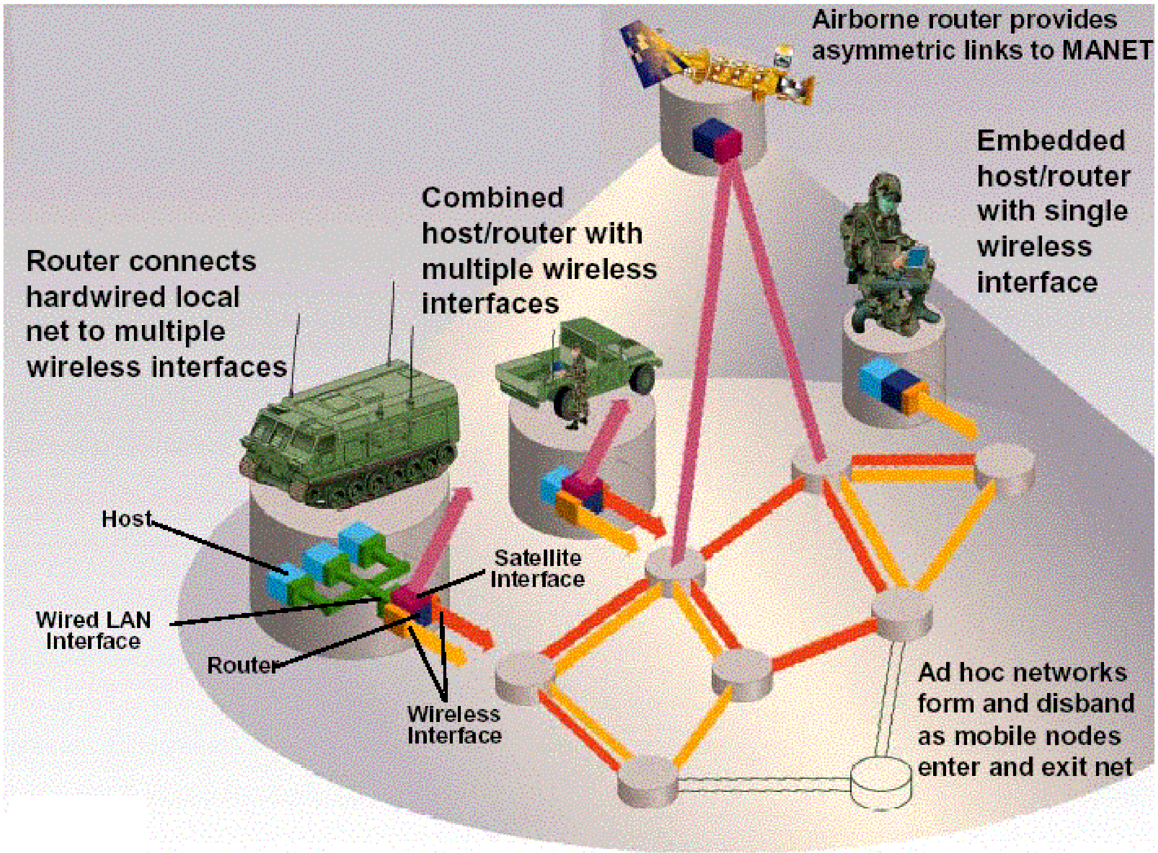

Figure 2.4. A typical MANET [from Corson 02].

2.3 Problems

As a MANET is made of highly mobile nodes, transmission of data from the source to the destination is generally done very inefficiently. They are many clustering and routing algorithms designed for cellular and wired networks. These algorithms have been modified for MANETs. Most of these algorithms haven't been successful in addressing the problems that affect MANETs. Power and security are the two important factors that play a key role in MANET and are considered to be the biggest challenges for MANETs.

Nodes present in MANETs are electronic devices that are able to transmit and receive messages. A battery is most common source of power in a MANET node. The batteries used by these mobile nodes are very small in size and have limited capacity. The power of a battery is expended whenever the node listens, transmits, receives or computes data. Since a node also routes messages of other nodes, they are high chances of the node's battery power being exhausted. This will affect the network survivability.

Due to the infrastructureless nature of MANETS, there is not fixed or central point in the network to ensure security. Wireless MANETs therefore have more security issues than the conventional wired and wireless networks. Some of the possible link attacks range from passive eavesdropping to active interference. An attacker can compromise, hijack or even capture a mobile node resulting in data being revealed or corrupted. Furthermore, a compromised node may be sued as a zombie to attack other nodes. A compromised node can listen to and modify all the traffic on the communication channel and may pretend to be a valid user. Finally the attacker may try to flood the network with packets leading to wastage of power resources of the network and frequent disconnections in network resulting in reduced survivability of the network. Availability, confidentiality and integrity are therefore important security aspects of a MANET. Availability ensures the survivability of the network despite attacks, while confidentiality ensures that certain information is never disclosed to unauthorized entities and integrity guarantees that a message being transferred is never corrupted.

Multicasting provision is another problem facing ad hoc networks. Researchers have been unable to come up with suitable protocols that can provide multicasting in an effective manner. Multicasting will help ad hoc networks to provide services like video conferencing etc.

CHAPTER III

CLUSTERING IN MANET

The concept of dividing a geographical region to be covered into small regions is defined as clustering. Each cluster will be uniquely identified using its cluster head. A cluster head is a node present in the geographical region of the cluster and takes care of the routing and allocation of resources for the cluster members. Nodes that have registered with the cluster head become members of the cluster. Clusters may change dynamically, reflecting the mobility of the underlying network. Some of the clustering algorithms are explained below. Most algorithms assume that all the nodes present in the network are same, that is have the same power capacity, transmission range etc.

3.1 Simple Clustering

Some of the simple clustering methods for MANETs are given in this section. These are considered to be the basic clustering methods.

3.1.1 Lowest ID Clustering

Each node present in the network is given a unique ID. All the nodes communicate with their neighbors using beacons and piggybacking their ID. A node, which hears from nodes having ID value higher than itself, is a cluster head. A node can be part of multiple clusters. These nodes are called gateway nodes [Gerla and Tsai 95].

3.1.2 Connectivity Clustering

In this method all nodes broadcast the list of nodes they are able to hear from. A node that is connected to most of the highly connected uncovered neighbor nodes is elected as the cluster head. A node is said to be uncovered if it does not belong to any cluster. If more than one node has the same degree of connectivity then the decision is made on the basis of Lowest ID. Both these methods belong to single hop clustering algorithms, where any two nodes are at most two hops away and cluster heads are not directly linked [Gerla and Tsai 95].

3.1.3 Cell Clustering

This technique is used in Mobile IP, here a region is divided into cells and a group of cells are clustered. When we have lot of hand off between two cells then they are brought under one cluster.

3.1.4 Weighted Clustering

In this method, each node will calculate its weight based on some of its characteristics. The node with the highest weight is elected as the cluster head of that cluster [Chatterjee et al. 02]. Expression for weight is given as:

- Weight = a*speed + b * degree + c *power + d *energy-left.

Where a, b, c, d are positive or negative depending on circumstances.

3.2 Enhanced Clustering

This section describes some of the latest and more complex clustering techniques used in MANETs.

3.2.1 K-Cluster approach

In this approach, a cluster is considered to be a subset of nodes, which are mutually reachable by a path of length at most K, for some fixed K. This is a graph based approach, where the network is considered to be a whole single connected graph. There is a path from each node to every other node through the edges of the cluster in the graph. Each node maintains three data structure tables containing the information of neighbor nodes, all the clusters and designated boundary nodes present in the network. Gateway nodes are called as the boundary nodes and one among these is given the designated boundary node status. The events that take place when a new node joins the network are given as follows [Chen et al. 02] [Kim et al. 98].

- Node sends messages to the neighbors.
- Each neighbor sends list of its neighbors and cluster list to the node.
- Determines already present cluster in the cluster set of neighbors and stores them in local list, which is a temporary list.
- Node try's to create new cluster by including the neighbor nodes and checking to include the neighbors of those nodes.
- Finds the essential clusters and assigns them new ids.

- Appends the essential clusters to local list and eliminates the redundant clusters in the local list.
- Determines the new boundary nodes from the updated cluster list and broadcasts them to the neighbors along with the cluster list.
- When a node disappears from the network then it is detected by its neighbors and the update procedure is initiated only by the cluster mates of the node.

3.2.2 Hierarchical Clustering

In this graph scheme, all clusters are considered to be connected and have minimum and maximum size limitations. Some of the other constraints placed are two clusters should have low overlap and clusters should be stable across node mobility. The clustering algorithm consists of two parts [Sucec and Marsic 02] [Banerjee and Khuller 01].

- Tree Discovery: a node in the sub-graph will initiate the spanning tree process, which is implemented using the breadth first search in the post order. Each node chooses its parents in the tree based on the shortest distance to the root.
- Cluster Formation: When a node detects its sub tree size has crossed the maximum size, it will initiate the cluster formation of its sub tree. Also when the node detects that the clustering is of poor quality then re-clustering is scheduled.

3.2.3 Dominating Sets Clustering

In this graph scheme the clustering is done using the dominating nodes present in the graph. The dominating set for a graph G= (V, E) is defined as a subset $S \leq V$, such that every vertex $U \in V$ is either in S or adjacent to a vertex of S. Three colors white, gray, black are used to classify the nodes present. Initially all the nodes are colored as white and when the node is changed to black all its neighbors are changed to gray. The black nodes form the dominating set and each vertex present in the dominating set is the cluster head [Chen and Liestman 02].

3.2.4 Max-Min D Clustering

The approach has the following data structures and functions defined [Amis et al. 00].

- WINNER: after the cluster head is selected, the winning node id of a particular round is used to determine the shortest path back to the cluster head.
- SENDER: is the node that sent the winning node id, as in WINNER.
- Floodmax function: each node locally broadcasts its WINNER value to all its one hop neighbors for a given round. The largest of the WINNER values is the selected as the new WINNER value for all the nodes. This process is continued for d rounds.
- Floodmin function: this is similar to Floodmax and also lasts for d rounds. Except a node selects the smallest value as its WINNER instead of the largest value.

- Overtake function: during flooding the WINNER values are propagated to neighbor nodes. Overtaking is the process of selecting a new value different from the node's own id, based on the outcome of information exchange.
- Node Pairs: a node pair is any node id that occurs at least once as a WINNER in both the first Floodmax and second Floodmin d rounds of flooding, for an individual node.

In the first stage the largest node id in each node's d-neighborhood is propagated using the d rounds of Floodmax. Nodes record their winning node for each round and at the end of Floodmax the surviving nodes are elected as cluster heads in the network. In the second stage d round of Floodmin is used to propagate the smaller ids that have not been overtaken. At the conclusion of the Floodmin, each node evaluates the round's WINNER to best determine their cluster head. The following rules are given for the cluster head selection criteria.

- Rule 1: if a node has received its own id in second round of flooding then it can declare itself as the cluster head and skip the rest of the process or continue.
- Rule 2: Once a node has identified all the node pairs, it selects the minimum node pair to be the cluster head. If a node pair does not exist for a node then proceed to rule 3.
- Rule 3: Elect the maximum node id in the first d rounds of flooding as the cluster head for this node.

After determining the cluster head, the node informs the cluster head about its membership. If there are neighboring nodes with cluster head selections that are different then these nodes are called as gateway nodes.

They are many other clustering algorithms that are based on the above approaches, and try to improve the efficiency of clustering.

CHAPTER IV

ROUTING IN MANET

Routing is the process of sending data in a particular path or paths from the source to destination. Since the advent of packet radio networks, numerous protocols have been developed for ad hoc networks. The issues that are to be considered during the design of the routing protocols are the typical limitations of the network like high power consumption, low bandwidth and high error rate. The current protocols may be generally categorized as Table driven and Source initiated or demand driven routing [Royer and Toh 99].

4.1 Table-Driven Routing

In these routing protocols the up-to-date entire routing information is maintained by each node present in the network. In order to uphold a consistent view of network, the updated topology of network is propagated among the nodes and this information is stored using one or more tables. The protocols differ from each other in the number of tables required and how the changes in network topology are to be propagated [Royer and Toh 99].

4.1.1 Destination-Sequenced Distance-Vector Routing (DSDV)

This is an updated Bellman-Ford routing algorithm without the loops in the routing tables. The table contains all possible destinations in the network, from the node along with the number of hops to reach and each entry is marked with a sequence number. They are two types of packets for transmitting routing information. Full dump packet carries all available routing information and can require multiple network protocol data units. Increment packets are used to relay changes in topology after a full dump. The route labeled with the most recent sequence number is used for routing data.

4.1.2 Clusterhead Gateway Switch Routing (CGSR)

A node is elected as the cluster head using a distributed cluster head selection algorithm, for a group of nodes that form a cluster. CGSR uses a modified version of DSDV in order to utilize the elected cluster heads. It also uses gateway nodes that are in more than one cluster. A packet sent by a node is routed through its cluster head to other cluster heads and gateway nodes present in the path, until the packet reaches the destination node cluster head. In addition to the routing table in DSDV the CGSR nodes maintain a cluster member table that is broadcasted periodically to all the nodes in the network [Krishna et al. 97].

4.1.3 Wireless Routing Protocol (WRP)

The goal of this protocol is to maintain routing information among all nodes in the network. Each node maintains four tables. They are Distance, Routing, Link-cost and Message retransmission list (MRL) table. To ensure connectivity each node must send hello message within a specified period of time. When a node receives a hello message from a new node then the new node is added to the routing table and a copy of routing table information is sent to it. The update message is used between the mobiles to inform link changes. The update message contains updates of the distance to the destination, the predecessor, list of response indicating the mobiles that should give acknowledgment. A mobile sends update messages to only its neighbors after processing updates from neighbors or detecting a change in a link to neighbor.

4.2 Source-Initiated On-Demand Routing

The routes are discovered only when the source node request for a path to the destination. A node that requires route to destination initiates a route discovery process, that is completed once a route is found or all possible route permutations have been examined. The route maintenance procedure maintains the route until the destination becomes inaccessible along every path from the source or until the route is no longer desired. Some of the source initiated routing protocols are given as follows [Royer and Toh 99].

4.2.1 Ad Hoc On-Demand Distance Vector Routing (AODV)

This algorithm is built on the DSDV algorithm. Instead of maintaining a complete list of routes the AODV creates routes on demand basis. It broadcasts a route request (RREQ) packet to its neighbors, who then forward request to their neighbors until the destination or an intermediate node with fresh route is located. When the RREQ packet is forwarded, intermediate nodes record in their route tables the address of the neighbor from which it received the broadcast packet. Once the RREQ packets reach the destination or intermediate node with fresh route a unicast route reply (RREP) packet is sent back. Destination sequence numbers are used to ensure all routes are loop-free and contain the most recent route information. If the source node moves then it initiates a new route discovery protocol to destination and if an intermediate path node moves then a link failure notification message is propagated to upstream neighbor. Even hello messages are used to maintain the local connectivity among nodes.

4.2.2 Dynamic Source Routing (DSR)

A route cache is maintained by each mobile node to store the source routes that are continuously updated as new routes are learned. The route discovery and route maintenance are the two major phases. A mobile node checks for the unexpired routes in the route cache in order to send the packets. If it does not have any routes it initiates the route discovery by broadcasting the route request packet. Each node receiving the packet checks in its route cache. If it does not find any route it adds its own address to the route record of the packet and forwards the packet along its outgoing links. When the packet reaches the destination, the route record contains the sequence of hops from source to destination. Route error packets and acknowledgments are used in the route maintenance phase [Kim et al. 98].

4.2.3 Temporally Ordered Routing Algorithm (TORA)

The algorithm is proposed for a highly dynamic mobile networking environment and is a highly adaptive loop-free distributed routing algorithm based on the concept of link reversal. The process is initiated by the source node and provides multiple routes for any desired source and destination pair. Localization of control messages to a very small set of nodes near the occurrence of a topological change is the key design concept of TORA. The protocol has Route creation, Route maintenance and Route erasure as the basic functions. There is a potential for oscillation and instability similar to count-to-infinity problem in distance-vector routing, to occur in TORA.

4.2.4 Associativity-Based Routing (ABR)

The protocol is free from loops, deadlocks and packet duplicates. According to this protocol a route is selected based on the degree of association stability of mobile nodes. Association stability is defined by connection stability of one node with respect to another node over time and space. When a node receives beacon from neighboring node, then the associativity tick of the current node with respect to the beaconing node is incremented. Associativity ticks are reset when the neighbors of a node or the node itself moves out of proximity. A high degree of association stability or connection stability indicates the two nodes have relatively low state of mobility, while a low degree indicates a relatively high state of mobility among the nodes. Route discovery, Route reconstruction and Route deletion are the three phases of ABR.

4.2.5 Signal Stability Routing (SSR)

The SSR selects routes based on the signal strength between nodes and a nodes' location stability. It is subdivided into two cooperating protocols Dynamic Routing Protocol (DRP) and Static Routing Protocol (SRP). All the transmissions are received and processed by the DRP into the Signal Stability Table (SST) and Routing Table (RT). The signal strength of neighboring nodes obtained from the periodic beacons is recorded in the SST. After updating the appropriate table entries the DRP passes a received packet to SRP, which processes packets. The SRP passes the packet up the stack if it is intended receiver or looking up the destination in RT and then forwarding the packet if it is not. An error message is sent to the source when a failed link is detected in the network, indicating the channel failed.

CHAPTER V

APPROACH

The proposed sector-based approach to clustering provides overlapping clusters. The multi-path routing algorithms will be able to utilize this overlapping cluster architecture and generate multiple paths for data transmission. In this chapter, the concept of sectors and node distribution value are introduced for clustering in order to facilitate load balancing on the cluster heads. The proposed approach also provides additional security to data transmitted between nodes which are at a higher security level. This is achieved by providing multiple cluster memberships to nodes requiring high security. The proposed routing approach provides multiple paths in a controlled way. In order to reduce intruder access, the routing algorithm uses a max-min strategy. In other words, the route generation starts at the destination instead of at the source node and the paths tend to be elliptical with source and destination at their foci. This increases the difficulty in predicting a node that belongs to a path used for data transmission. Moreover, the routing algorithm increases network survivability by using multiple paths for each transmission and increases the successful transmission rate by considering the association between the node and its neighbors.

We first define the following terms for Sector Based Clustering and Sector Based Routing algorithms.

Sector: The distance a node can communicate is the same in all directions and so the nodes communication region may be viewed as a circle with the node at its center. This circle is divided into sectors. In figure 5.1 a node is surrounded by 4 sectors. The number of sectors for a node is a variable for providing security and routing. The number of sectors is defined at initialization and can be even or odd.

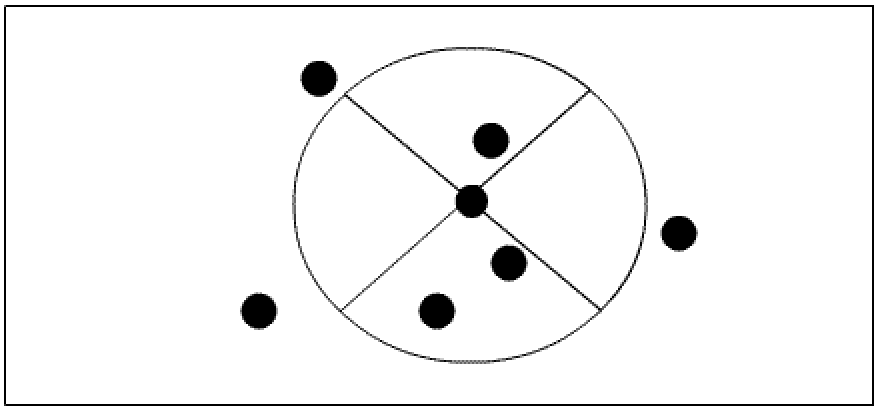

Figure 5.1.Sectors surrounding a node.

Security Level: All the nodes present in the network do not require the same security level. Depending on the security level desired by a node, each node is assigned a security level (L). For example, L=0 indicates a low security requirement and L=5 indicates a high security requirement.

Power Level (P): The power level is the power remaining in the battery of a node. A minimum power level is considered below which a node cannot act as a cluster head.

Adjacency value: This value indicates how long two nodes were adjacent to each other. Each node in the network keeps track of all the nodes that are present in its communication region. The number of clock ticks the two nodes were adjacent to each other is used for this purpose. If a node moves out of a neighboring node's communication region then the adjacency value corresponding to the node is reset.

Node Distribution Value (NDV): The distribution of nodes around a given node is measured by a node distribution value. The degree of a node only measures the number of nodes surrounding a given node, whereas the NDV indicates the spatial distribution of the nodes around a particular node. The higher the NDV, the more evenly the nodes are distributed in different sectors of that node. The NDV for a node is calculated as follows:

- Let each node have N equal sectors and each sector has a start and end angle (Figure 5.2).
- Based on the presence or absence of nodes in a sector, a weight is associated to each sector that is relatively opposite to a sector I (S_i). The weights are given based on the following rules:
    - θ of S_i is given as (start angle + end angle) / 2, θ can be considered clockwise or anti-clockwise but it should be consistent. Figure 5.2 has θ given in an anti-clockwise direction, from the 0 degree to the mid point of the sector.

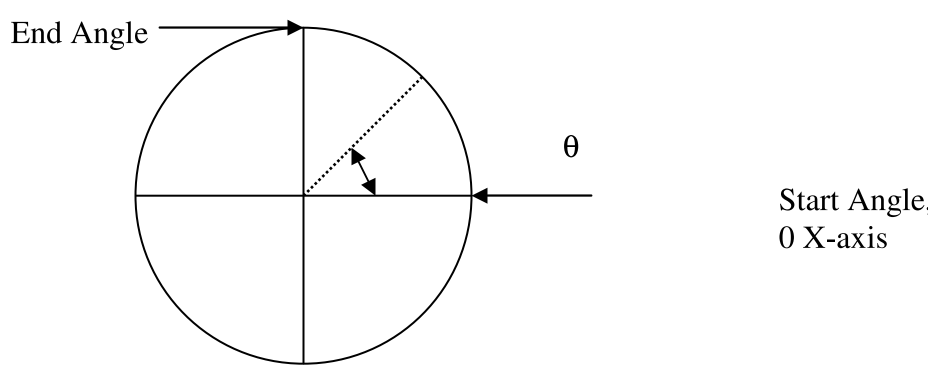

Figure 5.2 Node with 4 sectors and θ measured in an anti-clockwise direction.

- If $\theta + \pi \in S_k$ Then Weight (S_k) = $[N / 2]$ (Note: if measure in an anti-clockwise direction, then subtract, that is, $\theta - \pi \in S_k$ Then Weight (S_k) = $[N / 2]$)
- If $(\theta + \pi \pm (2\pi / N) * j \in S_k)$ and $(\theta + (\pi / 2) \leq (\theta + \pi \pm (2\pi / N) * j) \leq (\theta - (\pi / 2))$ Then Weight $(S_k) = [N / 2] - j$, where $[N / 2] \geq j \geq 0$. Here j determines the weight assigned to relatively opposite sector (figure 5.3 below). The maximum weight assigned is $[N/ 2]$ and the minimum is 0.
- The rest of the sectors not initialized above have Weight =0. These are sectors that are not relatively opposite to the sector S_i under consideration.

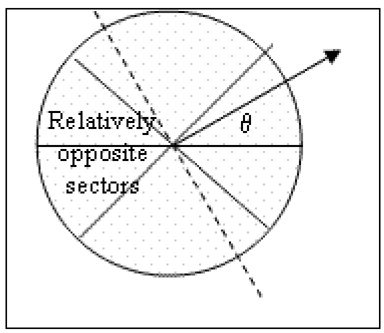

Figure 5.3. Relatively opposite sectors.

- The Distribution relative to S_i is given by

  $D(S_i) = \sum \text{Weight}(S_k)$,

  such that $i \neq k$ & S_k has at least one node in it.

- NDV for the node = $\sum D(S_i)$,

  such that S_i has at least one node in it.

Using the above procedure the NDV value calculated for the node shown in fig 5.1 is 4. There are two sectors that are opposite to each other and both these sectors contain nodes that neighbor the node under consideration. The maximum distribution relative to one of the two sectors is 4 / 2 = 2. Since both the sectors have a distribution value 2, and the other sectors contain no nodes, we have NDV for the node 2+0+2+0 = 4. The number of nodes present in a single sector does not affect the distribution value. The only criteria for associating a value to a sector is the presence of a node and the position of the sector with respect to the sector we are considering.

5.1 Sector Based Clustering

The Sector Based Clustering algorithm is divided into three stages:

Clusterhead Decision: In this stage each node evaluates itself to decide whether it is eligible to become a clusterhead. The decision will be based on the following conditions.

- Power level P of that node should be greater than the minimum acceptable power level. The minimum value is applicable for the entire network and ensures that the nodes do not die out early.
- NDV of node should be greater than N+1, where N is the number of sectors. This condition ensures that the cluster head is surrounded by neighboring nodes and is at the relative center of the cluster.

If the above conditions are satisfied then the node is eligible to become a clusterhead.

Member Inclusion: All the nodes that have evaluated themselves as eligible cluster heads will send membership invitations to other nodes to become part of their cluster. A node can accept to a maximum of M nodes in each sector, so there will be a maximum of NM nodes as members of a cluster. The value of M is fixed and the same for all sectors and nodes. This value may be decided based on the network density and power resources associated with nodes at network initialization. When a clusterhead has more than M nodes in a sector then it chooses the M nodes that have the highest values according to the following expression.

Adjacency value (node) + Security Level (node)

Consider the figure 5.4 below. Suppose node A has just come into the sector. Therefore B, C, D have a greater adjacency value. Assume security is same level. As node A is not accepted as a member of the cluster, it becomes a cluster head and nodes B, C, D become gateway nodes. If node A remains in the same region, it’s adjacency value will increase and it may eventually become a cluster member. This ensures that fast moving nodes which do not remain within an area for a significant amount of time do not alter the cluster configuration. This reduces the overhead for cluster formation.

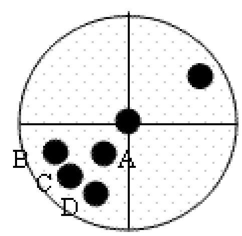

Figure 5.4. Example of cluster formation.

<u>Membership Decision:</u> Although the cluster heads cannot be members of other clusters, a node can be part of multiple clusters. The above conditions should always be satisfied. Furthermore, an upper limit is set for the number of clusters a node can be a member of based on its security level. If a node reaches its maximum number of clusters, it will reject additional membership invitations. When a node that is eligible to be a cluster head gets membership invitations then its decision is based on the following conditions.

- Number of invitations received is greater than the security Level of the node

- Security level of node is greater than the number of invitations sent to nodes that have higher security level than that of the node.

Based on the above two conditions a cluster head will be able to notice other cluster heads surrounding it and decide whether to continue as a cluster head or change its status. If the above two conditions are satisfied then the node does not continue as a clusterhead and accepts the membership of other clusters. If the conditions are not satisfied, it will reject the membership invitations and will continue as a cluster head. If a cluster head is currently transmitting data then it will delay its decision to change its status to a member or a gateway node. This will prevent transmission errors because of cluster head status change. The decision to change is implemented after the cluster head finishes data transmission. In the special case of a node that is not a member of any cluster, the node will act as a cluster head and invite other nodes to join as members of its cluster. Nodes that are part of multiple clusters are called gateway nodes as these nodes help in routing between cluster heads.

The cluster formation using the above three stages is applied to the set of nodes given in figure 5.5 below. Initially after all the nodes send their beacons to other nodes, each node calculates its node distribution value. The NDV is given in closed brackets under each corresponding node. Assume all the nodes have the same security level and sufficient power resources. Initially nodes having id's 0, 3, 6, 8, 9, 10, 12, 13, and 14 tend to be the cluster heads. As further interactions between these nodes take place nodes 0, 8, 10, and 14 prefer to be gateway nodes. The shaded nodes 3, 6, 9, 12, and 13 form the cluster heads. The figure 5.6 has the cluster formation for a similar network configuration considering the security of the nodes. The NDV value and security of the node are given respectively in closed brackets for each node. The nodes 2, 7, 9, and 12 have a high security requirement when compared with the rest of the nodes in the network. These nodes should be provided more security by including them in multiple clusters. The nodes that have lower security requirements tend to be cluster heads hence we have nodes 3, 6, 8, 10, and 13 as the cluster heads.

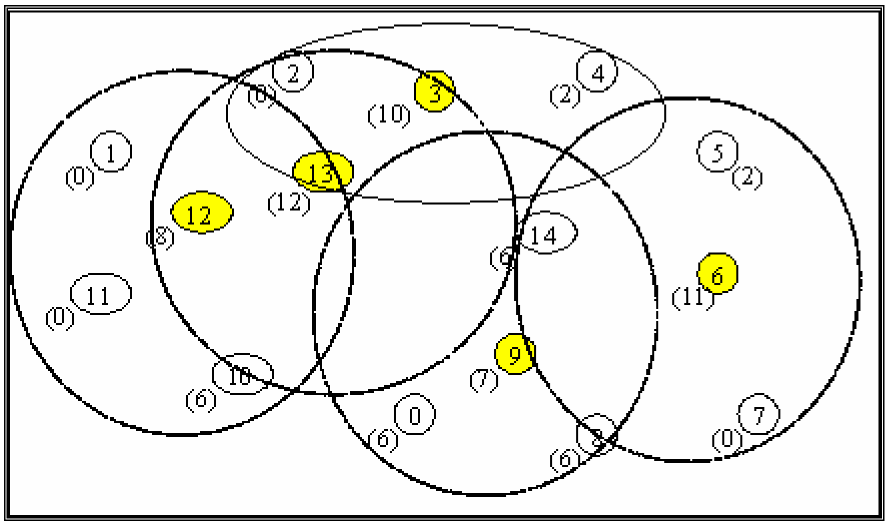

Figure 5.5. Cluster formation

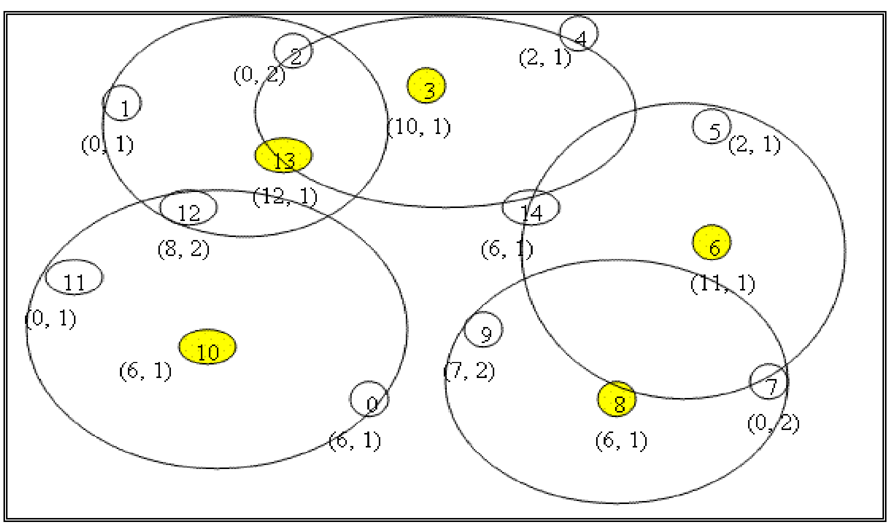

Figure 5.6. Cluster formations taking security into consideration

The number of sectors should be optimum for the network considering the node density in the network and power resources of the nodes. If they are too many sectors then the overhead associated in maintaining these sectors is high, but this can be traded with the security desired for the nodes in the network. The number of members in each sector should also be decided based on the above trade-off for an optimum value. If we have very few members in a sector then the number of clusters in the network will be very high resulting in high power consumption for the cluster head. This is because many nodes may become cluster heads resulting in increased power consumption in the network. If we have many nodes as members in a sector then total members in a cluster will be high and there will be a power drain on the cluster heads.

5.2 Sector Based Routing

The proposed above clustering method will provide the flexibility of having multiple routes between nodes. A stochastic routing algorithm will be able to route data securely between nodes. Sector based routing algorithm is a stochastic routing algorithm that is initiated by the source. A table called a Membership table is maintained by the clusterhead that contains the cluster member nodes, the sectors they belong to and the other cluster heads they are associated with. The source broadcasts a message to the destination indicating the amount of data to be transferred and the location of the source node. The broadcast is implemented by sending the messages only to the cluster heads and gateway nodes. The Route Request message (RREQ) message is used for this broadcast mechanism that is described in the following sub sections. The routing protocol is divided into two phases:

Route-Discovery: This is initiated by the destination node after receiving the message from the source. In this phase multiple routes in the form of Route Replies (RREP)s are discovered from the destination to the source and these routes have the following features:

- They are no common sub-paths among paths.
- No common node (except source and destination) on two separate paths
- No cycles in paths.

The route-discovery algorithm is given as follows:

- A: the set of nodes that are on the discovered routes contains destination D.
- Select nodes surrounding D on the following basis:
    - θ is the angle of direction of the source from destination. The destination node receives the initial RREQ message from the source node that contains the location of the source node. We assume that there is a coordinate system for defining locations. This can be implemented using a GPS on a sensor for example. This can be used to calculate the value of θ.
    - If D is a cluster head then select gateway nodes or cluster heads from sectors that are within this $\theta + (\pi / 2)$ to $\theta - (\pi / 2)$ range. The location of the source is included in the RREP messages. This is used to calculate the value of θ relative to the current location.
    - If D is a non clusterhead then select a cluster head from each sector that is within $\theta + (\pi / 2)$ to $\theta - (\pi / 2)$ range.

        - Selection is based on adjacency value, power remaining, and security level of node. The node with the maximum value for the below expression is selected.

          Adjacency Value (node) + Power (node) + Security Level (node)

    - Add these nodes surrounding D to A.
    - Loop for each node of path i
        - If the node is S, add the entire path to R or else continue.
        - Calculate θ from the current node to the source and repeat the above process of node selection with a modified range for sectors. If the distance covered is more than half then select the sectors lying in the range $\theta + K$ to $\theta - K$. where K is $2\pi / N$, N is number of sectors. The distance (measured by the number of hops reported by the RREQ message) between source and destination is obtained from the route request message received by the destination. Although this is the distance along one path only, we use this as the distance measure. When the distance covered during route selection is less than half then the number of sectors present is considered. The sectors range is $(\theta + K * T)$ to $(\theta - K * T)$, where T is $\lfloor N / 3 \rfloor$. This ensures that a node in the opposite direction is not selected. The figure 5.7 given below has the above description given pictorially. The source node (S) considers all the sectors between the range $(\theta + K * T)$ to $(\theta - K * T)$ to select the next hop node in the route. The intermediate node (I) considers the sectors between the range $\theta + K$ to $\theta - K$ to select the next hop node in the route.

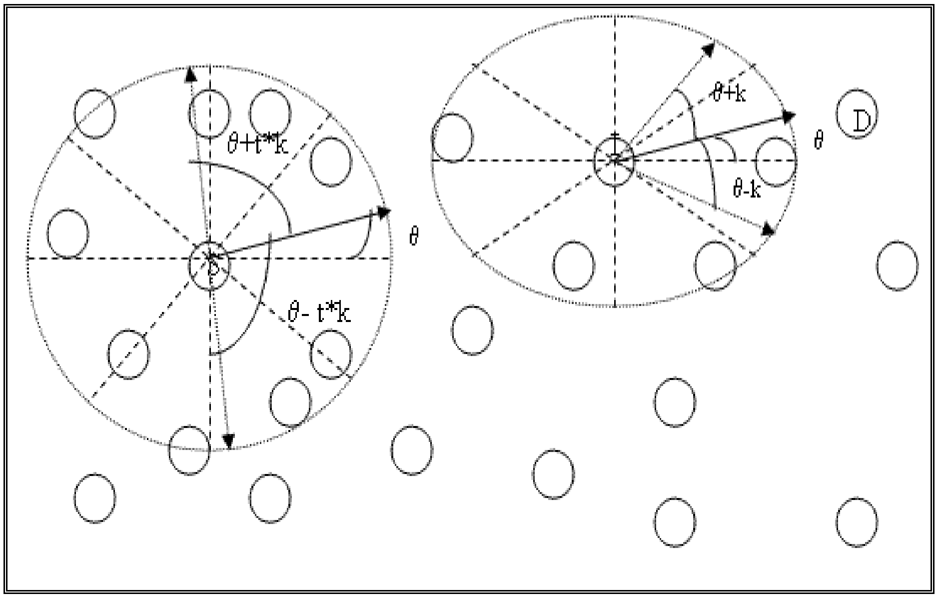

Figure 5.7. Route node selection considered sectors.

- If the node is already in A or there are no nodes, then abandon that path
- Connect the node to path i.

While the route information is being collected, the nodes are also selected based on their remaining power and adjacency value.

Initially when a source node needs to send data to some destination, it broadcasts a Route Request Message (RREQ) to all the cluster heads and gateway nodes surrounding it. This takes place before the route discovery process described above. The nodes broadcast the RREQ packet to all gateway or cluster head in a sector if the number of sectors is less than 5 or the angle of sector is greater than or equal to $\pi/2$. As in this case the broadcast message may not reach the destination if it is passed to a single node. If the sector angle is less than $2\pi/5$ (or number of sectors is more than or equal to 5) then it forwards RREQ to only one node either a gateway or cluster head in each sector. When the destination receives the RREQ it uses the above route discovery process to send Route Reply Messages (RREP). The RREP messages collect the path information. When the source receives an RREP message it adds the path to its list of routes for the current destination. The route discovery process will provide multiple paths between the source and destination. The ideal routes discovered using route discovery process is shown in figure 5.8 below.

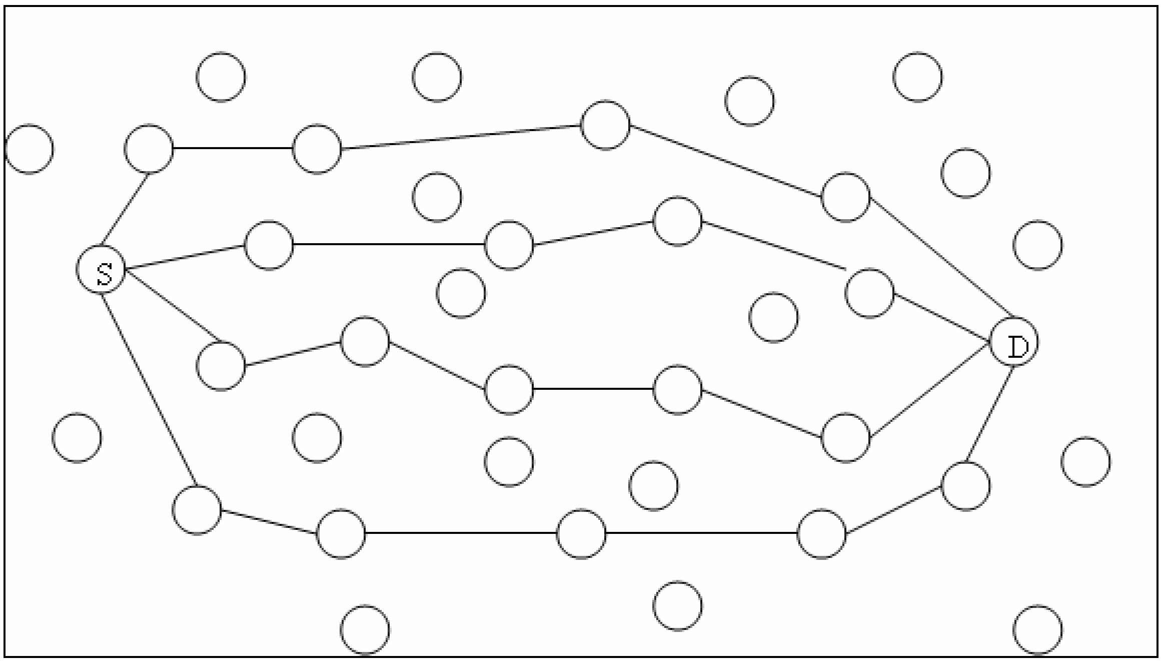

Figure 5.8 Routing paths in MANET using SBCR.

To ensure that the message is propagated along the network, multiple cluster heads or gateway nodes are chosen as the next hop in the path. A cluster head can communicate with nodes in its cluster including the gateway node. A cluster head can also communicate with another cluster head provided it is within range. A node that is not a cluster head or a gateway node can communicate with its own cluster head only. A node that is a gateway node can communicate with its own multiple cluster head. Depending on the purpose of the communication, a gateway node may not forward a message to all it associated cluster heads.

Data Transmission: Once the route discovery process has identified the routes the data transmission process is initiated. The amount of data to be transferred in a single attempt is regulated to a fixed size. The data is divided into small units and each unit is transferred using one of the routes discovered. A route i in R is selected with probability qi, where qi depends on the number of nodes in path i and its past selection. There are many possibilities for defining qi. The algorithm considered for qi is round-robin given as follows.

- The routes are identified using sequence numbers
- The route with the least sequence number is given the first preference.
- A route that was recently used will only be used again after all the other routes are utilized.

This gives higher probability to select routes that were not used for the longest period of time. If any of the nodes in a route is missing then that route is abandoned and an alternative route is picked. As we have multiple routes that are being used the power drain in a single path is reduced. Also since the paths are not necessarily the shortest path between source and destination it is difficult for a malicious intruder to predict the path that will be taken.

CHAPTER VI

SIMULATION

6.1 Objective

The objective of this simulation is to compare Sector Based Clustering and Routing (SBCR) with Ad hoc On-demand Distance Vector Routing (AODV) and Cluster Based Routing (CBR) protocol. The comparison is done with respect to the network survivability, that is, how long the network is able to survive. The simulation also considers the average number of available routes between source and destination (for SBCR only as AODV and CBR discover only single routes), the number of packets successfully transmitted from the source to the destination, and the overhead packets that are associated with the protocols for route establishment for all three protocols. The overhead packets and other protocol related definitions are given in the later sections.

6.2 Implementation Platform and Environment

The simulation was implemented on the OSU Computer Science Department's Sun Blade 150, which is a workstation-class computer. The system has 256 mega bytes of RAM. It also has 7.5 giga bytes of hard disk. It runs the Sun OS 5.9 operating system, which is a UNIX-based operating system.

6.3 Ad hoc On-demand Distance Vector Routing

The Ad hoc On Demand Distance Vector (AODV) routing algorithm [Perkins and Royer 99] is an on demand algorithm, in that it constructs routes between nodes only when desired by source nodes. The routes that are built are maintained as long as the source nodes require them. The routes are built using the route request / route reply query cycle. AODV constructs a single route between a source and a destination. The source node broadcasts a route request (RREQ) packet across the network when it desires a route to the destination. The intermediate nodes receiving the RREQ packet update the reverse route tables that point to the source node. The intermediate node may continue broadcasting the RREQ or will send a route reply (RREP) message. It sends a RREP message if it is either the destination or if it has a route to the destination with corresponding sequence number greater than or equal to that contained in the RREQ. A node will increment its sequence number if there is a change in the neighbor node configuration. The RREP message is propagated to the source node using the reverse table; also the node will set up forward pointers to the destination in the forward route table. If multiple RREQ with the same broadcast ID are received by a node then the message is discarded. Once the source node receives the RREP message it begins to forward data packets to the destination. After the transmission is completed the links will time out and eventually be deleted from the intermediate node routing tables. In case of a link break while the route is active the upstream neighbor node propagates a route error (RERR) message to the source node. When the source node receives the RERR it stops transmission and will reinitiate the route discovery if it desires a new route.

6.4 Cluster Based Routing

The Cluster Based Routing (CBR) [Krishna et al. 97] uses the lowest ID cluster algorithm. The node having the lowest id is considered to be the cluster head of a group of nodes in the same geographical region. In order to support the cluster formation a Neighbor table is maintained that contains the neighbor node information such as ID. When a node receives a beacon from a neighbor node then information corresponding to the neighbor node is updated. A node is considered to be in any of the three states Undecided, Clusterhead, or Member. When a node does not belong to any cluster it is considered to be in an undecided state. If a clusterhead detects a neighboring node that is also a clusterhead that has a lower ID then it changes its state to member otherwise it continues as clusterhead and the other node has to change its state to member. If a member looses its clusterhead status, it can change its state to clusterhead only if it has the lowest ID among neighbor nodes. If it does not have any nodes in communication range then it switches to an undecided state. In order to support the routing process, CBR uses a data structure called the cluster adjacency table (CAT). The CAT stores information of the neighboring clusters. The route discovery process is similar to the AODV route discovery process with the exception that the RREQ packets are broadcasted only to the Cluster heads using the gateway nodes. The protocol does not use any tables for maintaining the route information. When the RREQ packet reaches the destination it contains the loose source route with all the intermediate nodes in the transmission. The destination node sends a route reply message (RREP) to the source node using the reversed loose source route. Route error messages (RERR) are used to inform about the lost links. Like AODV, CBR discovers only a single route between source and destination.

6.5 Simulation Input Parameters

The input variables that are required for the simulation of the three protocols are given below.

Number of Nodes: The number of nodes present in the simulation environment. This is the number of nodes placed in random locations at the start of the simulation.

Boundary: The two dimensional boundary specifications within which the nodes move around. The maximum X and Y coordinates are requested from the user. The simulation ensures that the nodes do not cross these boundary specifications. The number of nodes and the boundary values determine the density of nodes in the network.

Speed: The maximum speed with which each node can move around in the simulation plane.

Power Level: The maximum power associated with each node. A node is active in the network until it is depleted of its power resources.

Radius: A node's radius of communication. This is the maximum distance a node will be able to communicate in any particular direction. The distance is equal in all the directions so the communication area is a circle with the node at its center.

Security Level: The maximum security level that can be assigned to a node in the network. The security level is assigned randomly to the nodes at the beginning of the simulation. A number between zero and the maximum security level as specified by the user is assigned as security level to a node.

Number of Sectors: The variable is specifically for SBCR. The number of sectors present around a node is decided before the start of the simulation.

Sector Nodes: This variable is also specifically for SBCR. The variable is the maximum number of nodes per sector a cluster head can have as members. The number of sectors and sector nodes set the upper limit for the number of members a cluster head can have.

Time: The maximum simulation time for each protocol.

6.6 Design of Simulator

Object Oriented Design (OOD) features were used in the design of the simulator. Each node is independent and has control over its mobility and data transmission. A three layer architecture as shown in fig 6.1 was used for the simulator design. The top most simulator interface layer interacts with the user and determines the specifications (see above sections) for the simulation. Its responsibility includes determining the simulation type the user requests and initiating them with the required constraints. The next layer is the protocol simulator, which is dependent on the protocol being simulated. It has an internal clock that keeps track of the simulation time. The clock is incremented after all the nodes are given opportunity to complete their tasks. Communication between the nodes is simulated using the network / transport feature present in this layer. When a node needs to send any message it will forward it to the network / transport layer that will handle the request. Data collection helps in collecting the statistical information regarding that simulation run. The last layer is the Node layer that helps in simulation of node behavior depending on the protocol. A random waypoint mobility model [Camp et al. 02] is implemented for this simulation. Nodes have the, capability to read the messages they receive and can transmit messages. Additional capabilities of a node depend on the protocol it is implementing. The initial node configuration is the same in the simulation of all the three protocols.

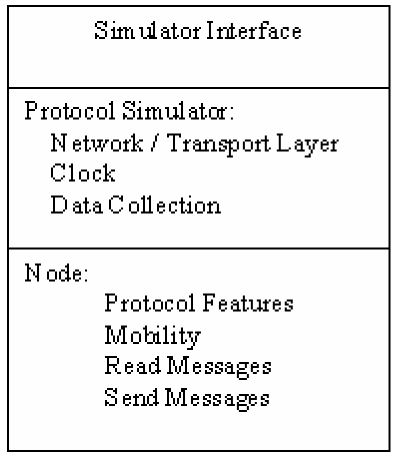

Figure 6.1 Simulator layers.

The mobility model considered for the node mobility is random waypoint mobility model. This model developed by Carnegie Melon University is widely used to evaluate Ad Hoc wireless networks protocols [Camp et al. 02]. For each node a random pause time is introduced in between each movement. The Maximum speed of a node, maximum pause time, number of nodes and simulation area are some of the inputs to this model. The mobility model for each node present in the network is as follows [Camp et al. 02]:

1. Initially each node is placed at a random location in the given simulation area.
2. For each node, a random speed is chosen that is less than the maximum speed and a random destination is chosen within the simulation area. The node begins to move towards the selected destination with the selected speed.

3. When the node reaches the destination, a random amount of pause time is chosen that is less than the maximum pause time. The node pauses at that location for the selected pause time.
4. The node repeats steps 2 and 3.

The basic node structure containing different variables to keep track of the nodes behavior is given in figure 6.2 below. Additional variables depending on the protocols are added to the derived structures that inherit this node structure.

```
class CNode
{
public:
      long iPowerlevel; //The power left in the node
      int iSecuritylevel; //The security level of the node.
      int iSpeed; //The speed with which it is traveling now.
      int iCurrloc_X; //The X-coordinate current location.
      int iCurrloc_Y; //The Y-coordinate current location.
      int iFutloc_X; //The X-coordinate future location.
      int iFutloc_Y; //The Y-coordinate future location.
      int iBoundary_X; //The X-coordinate boundary value.
      int iBoundary_Y; //The Y-coordinate boundary value.
      int iMaxspeed; //Maximum speed with which node can travel.
      bool bMobility; //Node is currently in motion.
      long iBeacontime; //Beacon interval
      long iLastbeacon; //When last beacon was sent.
      int iNodeid; //The node id in the network.
      long iPausetime; //The interval before it starts motion again
      //after reaching destination.
};
```

Figure 6.2 Node structure.

The three protocol simulators use tables to store the routing and protocol information. The tables are updated regularly based on the state of the network and timed out entries are constantly removed to keep the tables updated. Some of the individual tables with respect to the protocols are given below.

6.6.1 AODV Protocol Tables

Since AODV stores the routing information in tables it requires additional tables when compared to the other two protocols.

Reverse route: This table has an entry when a node receives an RREQ from another node. Using this table the source node can be easily traced. The structure of the table is given in figure 6.3.

```
class CReverseroute
{
public:
   int iSourceid; //The source node id
   int iSequencenumber; //Source node sequence number.
   int iNumberofhops; //Number of hops away
   int iNeighborid; //Next neighbor node to contact
   long iLifetimestamp; //Time since used.
   int iRreqid; //Route request id.

};
```

Figure 6.3 Reverse route AODV table structure.

Forward route: When a node receives RREP message from another node an entry is added to this table. The entries in this table help in forwarding messages to the destination node. The variables of this table are given in figure 6.4.

```
class CForwardroute
{
public:
      int iDestinationid; //Destination id
      int iDestinsequenceno; //Destination sequence number.
      int iNeighborid; //Next Neighbor id
      int iNumberofhops; // Number of hops remaining
      long iLifetimestamp; // Time stamp of last used.

};
```

Figure 6.4 Forward route AODV table structure.

Message table: All the messages a node receives are stored in this list. A node reads all the messages present in this list and acts accordingly for each message. The data structure designed for this list is given in figure 6.5 below. This data structure is also used to communicate between the layers in order to simulate communication in the network.

```
class CAodvmessage
{
public:
      int iMessagetype; //The message type 1-RREQ 2-RREP 3-RRER 4-Data
      0-Beacon.
      int iSourceid; //source node id from which message originated.
      int iDestinationid; //Node message should reach
      int iSourcesequenceno; //source node sequence no
      int iDestinationsequenceno; //Destination sequence no
      int iMessageid; //The message id used generally for RREQ and
      Data.
      int iForwardnodeid; //Node that forwarded the message.
      int iNodeforwardtoid; //Node to which message should be
      forwarded.
      long iTimetolive; //Time stamp.
      int iHopcount; //Number of hops.
};
```

Figure 6.5 AODV message structure.

Adjacency table: This has information about the neighboring nodes. When a node receives a beacon from a neighbor node it adds an entry or updates the table. If the node does not receive beacon from a neighbor for period of time then that neighbor's entry is removed from the table. The Adjacency structure is given in figure 6.6 below.

```
class CAdjacency
{
public:
      int iNeighborid; //Neighbor id
      int iNeighsequence; //Neighbor sequence number.
      long iLifetime; //Time stamp.
};
```

Figure 6.6 AODV adjacency structure.

6.6.2 CBR Protocol Tables

CBR uses adjacency and message tables. In the CBR protocol the route information is stored in messages. There is therefore no need for tables to store routing information.

Adjacency table: The use of adjacency table is similar to that in AODV. The structure of the adjacency table is different from AODV as it requires variables to store the cluster information. The structure is given in figure 6.7 below.

```
class CAdjacency_cbr
{
public:
      int iNeighid; //Neighbor id
      int iClusterid; //Cluster it belongs to.
      int iRole; //Role of the Neighbor 1-Clusterhead, 2-Member 3-
      Gateway.
      long iTimestamp; //Time stamp.
};
```

Figure 6.7 CBR Adjacency structure.

Message table: The purpose of a message table is similar to AODV, but the structure is different from AODV. It includes a list feature that can store the entire current route information. The data structure used for this table is given in figure 6.8 below.

```
class CMessage_cbr
{
public:
      int iMessageid; //Message id- for RREQ and Data
      int iMessagetype; //Message type 0-Beacon 1-RREQ 2-RREP 3-RERR 4-
Data.
      int iSenderid; //Message forwarded by node.
      int iSourceid; //Source node -Message originated from.
      int iReceiverid; //Message forwarded to.
      int iDestinationid; //Destination node.
      long iTimestamp; //Time stamp.
      vector <int> listRoute; //Route information.
      int iLocroute;//Current position in route.

};
```

Figure 6.8 CBR message structure.

6.6.3 SBCR Protocol Tables

Similar to CBR, SBCR uses adjacency and message tables. Moreover the routing information is stored in each message for SBCR. SBCR also uses a member table for the cluster heads to store the member information and gateway nodes to store their clusters information such as the clusterhead and the sector it belong to etc.

Adjacency table: In addition to the CBR adjacency structure the structure used here has the variables to capture the sector location and security level. Figure 6.9 below describes the data structure.

```
class CAdjacency_sbcr
{
public:
      int iNeighborid; //Neighbor id
      int iRole; //Role of Neighbor
      long iTimestamp; //Time stamp
      int iAdjvalue; //Adjacency value
      int iSector; //Sector located in
      int iSecuritylevel; //Security level of neighbor.
};
```

Figure 6.9 SBCR adjacency structure.

Member table: This table is used by the cluster head to store the information of its cluster members. The gateway nodes also use this table to store the information of their cluster heads. The information is updated regularly and is used for data transmission purposes. The data structure is given in figure 6.10 below.

```
class CMember
{
public:
      int iMemberid; //Member id
      int iRole; //Member role
      int iSector; //Sector belongs to
      int iSecuritylevel; //Security level of node.

};
```

Figure 6.10 SBCR member structure.

Message table: The structure used here is similar to CBR and also stores the location of the source node. This is used for routing purposes for this protocol. It also lists the route ID to differentiate between multiple routes. The structure is given in figure 6.11 below.

```
class CMessage_sbcr
{
public:
      int iMessageid; //Message id.
      int iMessagetype; //Message type: 0-Beacon, 1-RREQ, 2-RREP, 3-
RERR, 4-Data.
      int iDestinationid;//Destination id.
      int iSourceid;//Source id.
      int iSenderid;//Forward node id.
      int iReceiverid;//Receiver id.
      long iTimestamp;// Time stamp.
      int iRouteid;//Route id.
      int iLocx; //X-Coordinate.
      int iLocy;//Y-Coordinate.
      vector <int> listRoute; //Route information.
      int iLocroute; //Current route location.
      int iNoofhops; //Number of hops.
};
```

Figure 6.11 SBCR message structure.

6.7 Implementation Details

The simulation of these protocols was implemented in C++ on the Oklahoma State University Computer Science Department's Sun Blade 150 machine running Sun OS 5.9 operating system. The simulator is implemented so that it is able to provide three different types of simulation results. Type 1 simulates the three protocols until a certain percentage of nodes present in their network are dead. In Type 2 the simulation is carried until a fixed clock time for each of the three protocols. Finally in the Type 3 only the SBCR protocol is simulated for different values of number of sectors.

During the simulation the following results are collected in the specified output file:

Survival Time: The time network was able to survive for each of the three protocols. The number of nodes alive defines the survivability of the network. This parameter differentiates the three protocols with respect to their simulation times in Type 1 and Type 3 simulations.

Nodes Dead: A node is 'dead' if it has remaining power level less than or equal to zero. This parameter is investigated in Type 2 and Type 3 simulations. It indicates how many nodes are dead in the network at the end of the simulation time.

Average Node Power: This is the average power level of nodes present in the network. It indicates the overall network power consumption.

Messages Sent: The number of data messages sent from the source to the destination once the route has been established. This value is cumulative for all the nodes.

Messages Reached: The number of data messages that reached the destination. This value is cumulative for all the nodes.

Message Success Ratio: This value is calculated by dividing the above two parameters. As the number of data messages sent using the three protocols may be different (due to differences in the simulation times) calculating this ratio is very important as it defines the efficiency of the protocols.

Overhead: This is the number of route request packets (RREQ) that were totally sent in order to determine the routes between the source and destination.

Transmission Attempts: The number of times a node sends data to the destination. This value may differ from each protocol because of their inherent differences. For example, the rate of nodes dying will vary in the different protocols resulting in different number of transmission attempts.

Overhead Ratio: This is obtained by dividing the above two parameters. It indicates the number of overhead packets transmitted for each transmission attempt.

Average Paths: The average number of routes provided between source and destination. This output parameter is calculated only for SBCR as AODV and CBR both generate only a single route.

The simulator was designed as a menu based application giving the user the ability to set the input parameters. Initially the user is requested the number of nodes for the simulation run followed by the boundary dimensions. The maximum number of security levels is entered next. Each node is assigned a security level by the simulator. The nodes communication radius is set by the user. The number of sectors associated with each node and the number of nodes a cluster head can have as members in a sector is defined. These two parameters are used for the SBCR protocol only. The maximum speed with which a node can travel and the amount of power a node initially has is input by the user. Each node is assigned this level of power. After the output file name is entered by the user he is given the option to choose the type of simulation he requires. If he chooses Type 1 then the percentage of 'dead' nodes required for the simulation to stop is requested. In the case of Type 2 and Type 3 simulations, the maximum simulation time for each of the protocols is requested from the user.

The nodes in the network consume power based on their activity. In this simulation a node is considered to consume 250 units of power for data transmission and 150 units of power for data reception [RoamAbout 96]. Similarly 75 units of power are required for overhead packets transmission and 50 units for overhead packet reception. In order for the beacons to be transmitted and received 5 units of battery power is consumed. These figures are based on [RoamAbout 96] and are scaled to reflect the proportional power consumption among the different activities. The power required for mobility and computation is not considered in this simulation. Only the battery power consumed for wireless transmission is considered. Other parameters that are given fixed values or ranges in the simulation are as given below:

Beacon Interval: The interval between two beacons sent by a node is 8 clock units if the node is stationary else it is 3 clock units. Also a period of 5 clock units is given as grace period to send beacons initially for the network to become stable.

Pause Interval: when the node reaches its destination a random value between 5 and 30 is assigned as the pause time that is used in the random waypoint mobility model.

Tables Update Value: if the node is stationary then for every 15 clock units its tables are updated. If the node is mobile then for every 10 clock units its tables are updated.

The results were obtained by averaging several simulation runs. These results are represented as graphs and are given in chapter VII.

CHAPTER VII

OBSERVATION

The simulation for each case was run many times and the averages of the runs obtained. The output for the simulations is given in appendix B. The values obtained in the output files are converted to tables given in appendix C. The graphs for the three types of simulations are given in this chapter.

7.1 Type I

The simulation is run for each of the three protocols until a percentage of the nodes are dead. Simulations were conducted for different node densities. The number of nodes in the network was increased from 30 to 150 with increments of 10 in order to capture the behavior of protocols with respect to increase in node density. Results were collected for simulations after 5%, 10% and 25% of the nodes in the network are dead. All the graphs in this section are plotted with number of nodes in the network on the X-axis. The input values kept fixed for all these simulations are the boundary dimensions (1000 by 1000). The Maximum security level is set to 2 (levels 0, 1, 2) and node communication radius is restricted to 200. For SBCR, The number of sectors is 6 and the nodes per sector are 2. The maximum node speed is 3 and the initial power level is 100000 units.

7.1.1 Network Survival Time

Network Survival Time is the time the network is able to survive before certain percentages of the nodes are dead. The graphs given in figure 7.1, 7.2, and 7.3 represent 5%, 10% and 25% of nodes dead respectively. The corresponding tables for these graphs are given in appendix C tables C.1, C.2 and C.3. In all three cases SBCR clearly outperforms the AODV and CBR protocols. As the node density increases the time required for a certain percentage of nodes to die decreases. An Increase in node density increases the overhead packets. Furthermore certain nodes have a lot of overhead packets communicating through them and these nodes die soon. This explains the decrease in survival time with increase in node density. The decrease in survival time is less in SBCR compared to the other two protocols. This is because SBCR uses multiple routes between source and destination. AODV and CBR use a single and generally the shortest path between source and destination causing power drain in this path. These results show that SBCR route discovery mechanism provides routes with high security and multiple paths.

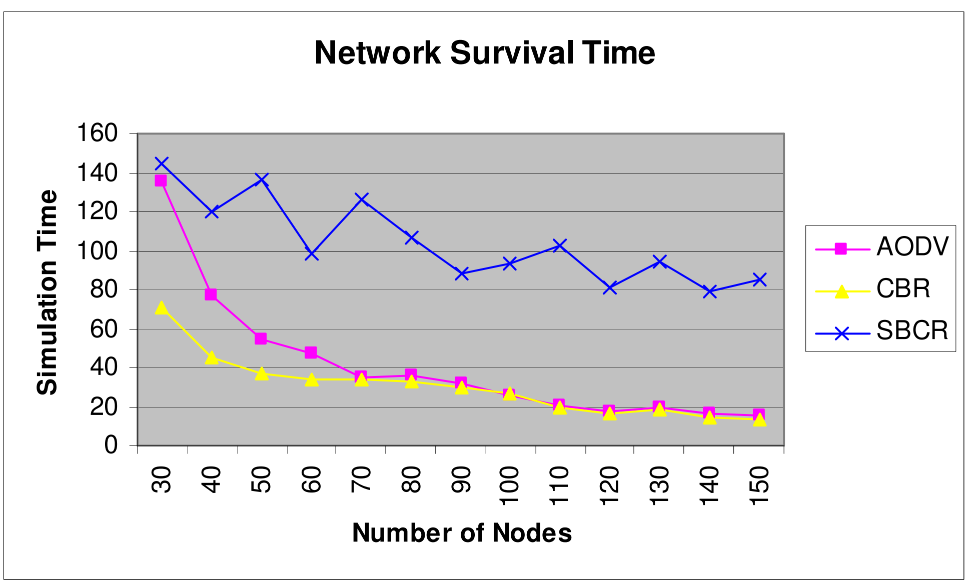

Figure 7.1. Network survival time with 5% of nodes dead.

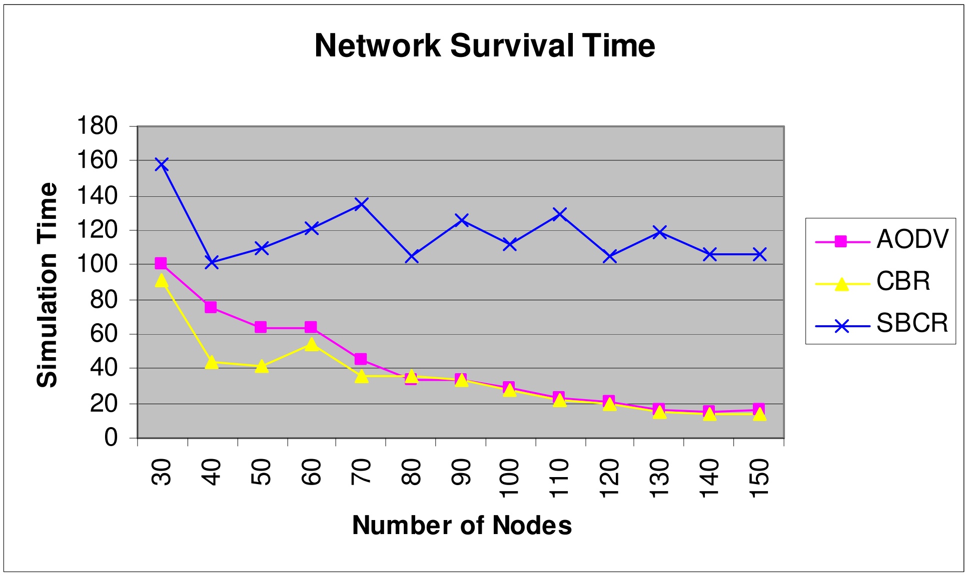

Figure 7.2. Network survival time with 10% of nodes dead.

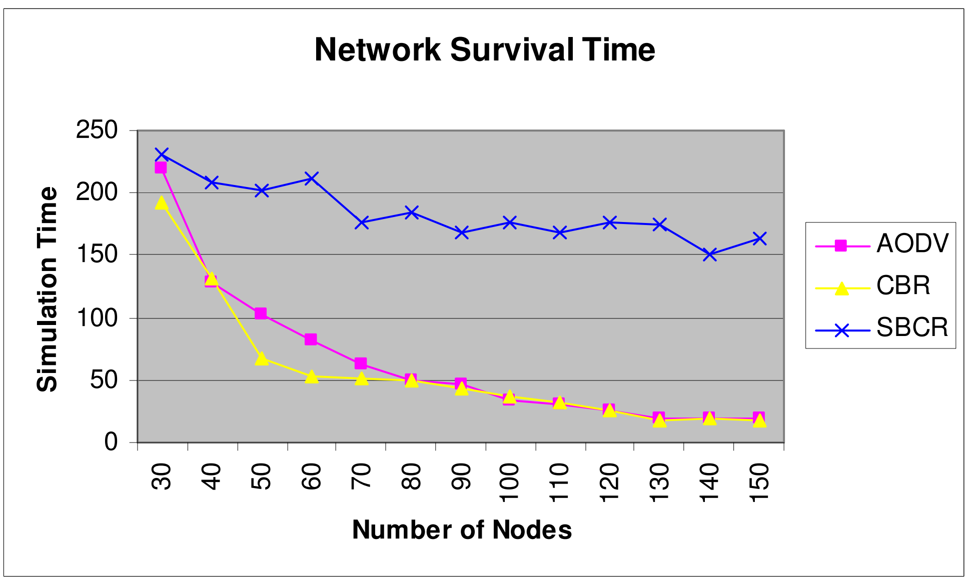

Figure 7.3. Network survival time with 25% of nodes dead.

7.1.2 Average Node Power

This defined the average power left in a node after simulation. The tables for the graphs of figures 7.4, 7.5, and 7.6 are - given in appendix C tables C.4, C.5, and C.6. In all the three protocols, the average power remaining decreases with node density. When the node density is less, the average power remaining for SBCR is less than for the other protocols because SBCR has a higher survivability. However, at higher node densities, the number of overhead packets is slightly less for SBCR compared to the other protocols. The number of transmission protocols is high for all the three protocols. The SBCR power consumption is more evenly spread among the nodes. Therefore the average remaining power is more for the proposed SBCR protocol. As the node density increases SBCR has better average power compared to AODV and CBR. The increase in node density also increases the number of overhead packets so the power levels decrease.

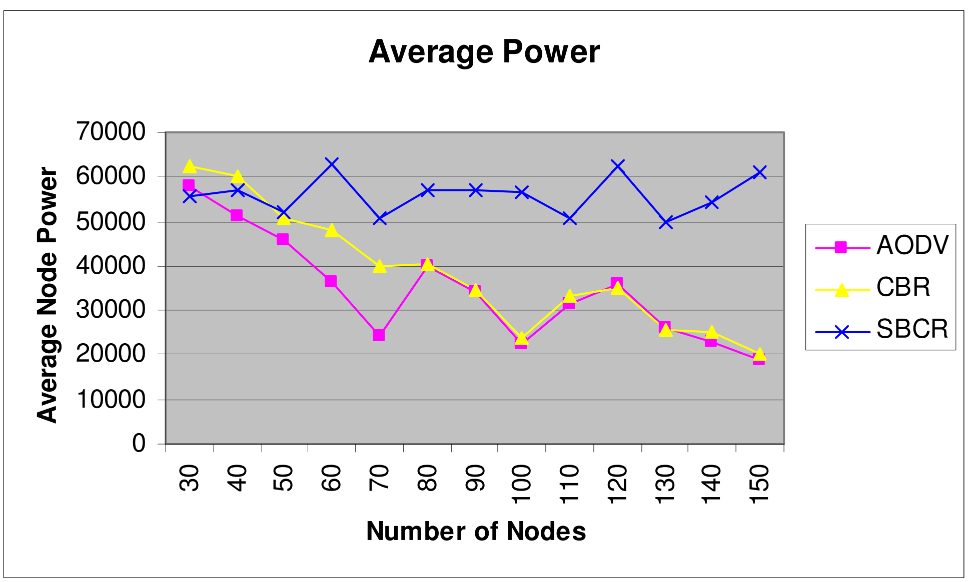

Figure 7.4. Average power left- 5% nodes dead.

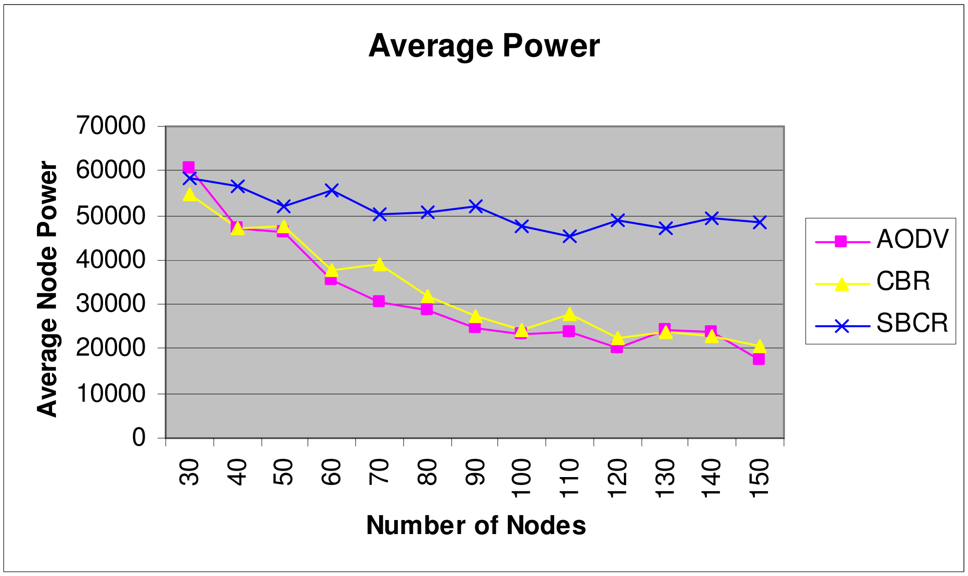

Figure 7.5. Average power left – 10% nodes dead.

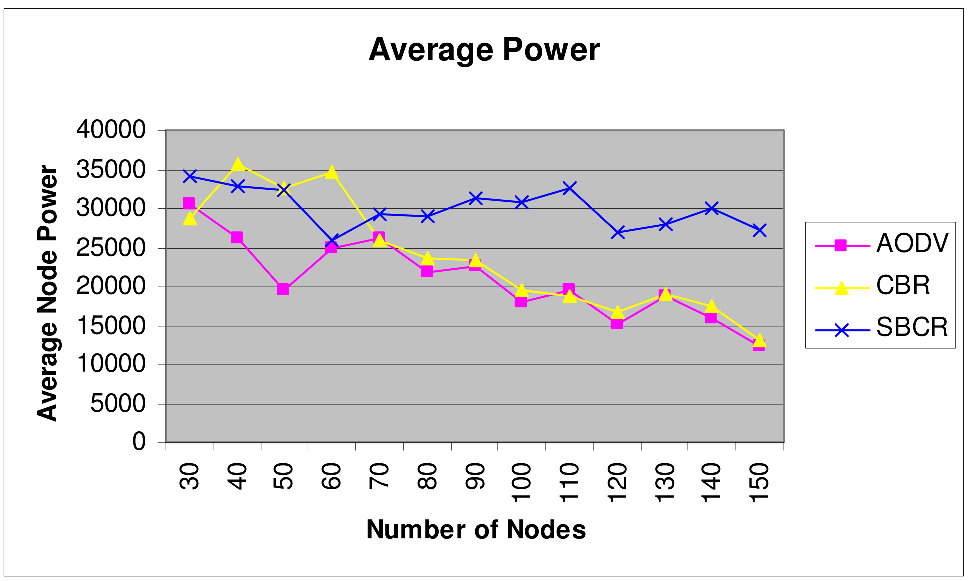

Figure 7.6. Average power left – 25% of nodes dead.

7.1.3 Message Success Ratio

This is the percentage of messages that successfully reached the destination node. This is obtained by taking the ratios of the total messages that were sent by the source nodes per simulation and the total messages that reached their destinations. The graphs given in figures 7.7, 7.8, and 7.9 are plotted using the values given in appendix C tables C.7, C.8 and C.9 respectively. In all the three graphs SBCR performs better than AODV and CBR. The problem with AODV is that it has a tendency to use already existing routes. As this is a very mobile network and connections between nodes are lost because of mobility, the loss of one link may cause a number of existing routes to be useless. Moreover before the source detects the loss of a route it would have transmitted many data packets. In the case of CBR the shortest path between the source and destination is used. The link may be lost because of power drain or node mobility. As SBCR has multiple routes and most of its routes are highly reliable the success ratio is very high.

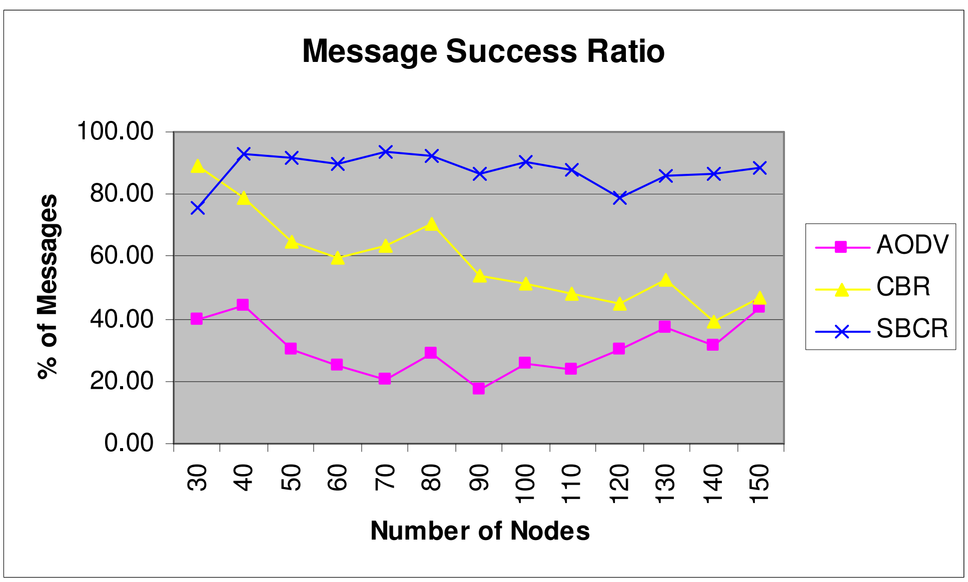

Figure 7.7. Message success ratios – 5% of nodes dead.

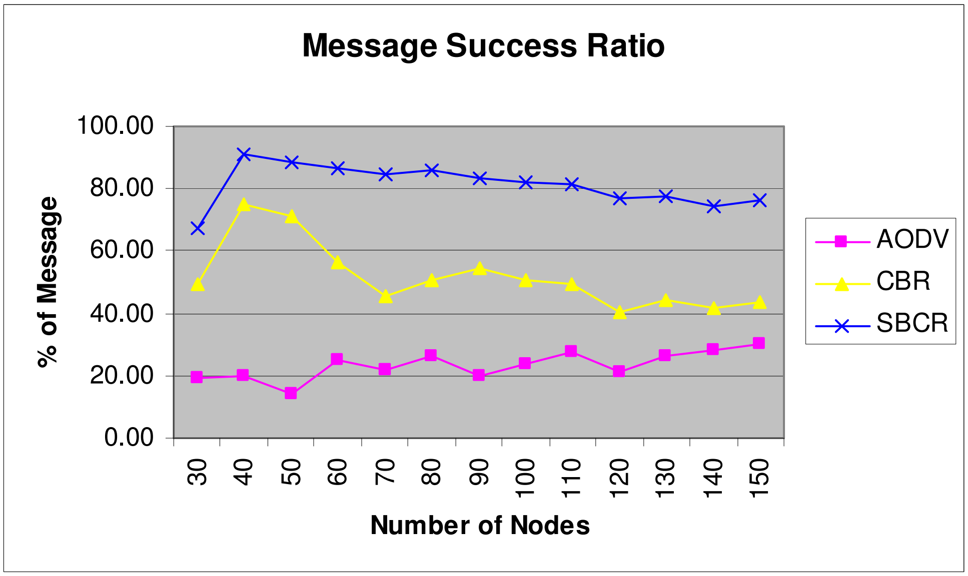

Figure 7.8. Message success ratios – 10% of nodes dead.

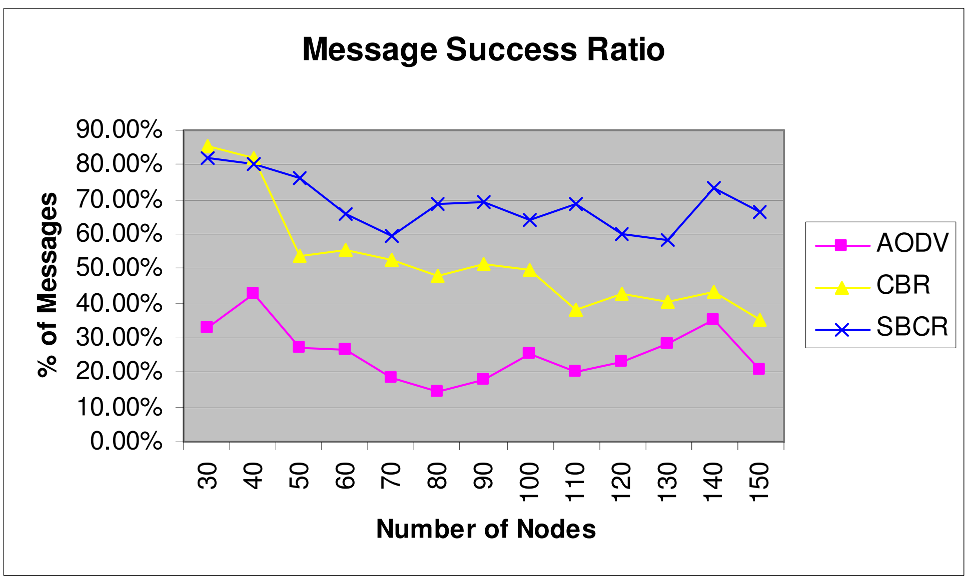

Figure 7.9. Message success ratios – 25% of nodes dead.

7.1.4 Overhead Ratio

This is the overhead incurred in order to discover a destination in the network. The graph given in figures 7.10, 7.11, and 7.12 show the overhead ratio with the number of nodes on the X-axis. These graphs are plotted from the values given in tables C.10, C.11, and C.12 in appendix C. In all the graphs and for all the three protocols the overhead ratio increases with the number of nodes in the network. As the node density increases the number of neighboring nodes increase and hence more overhead packets will be transmitted. This is particularly observed in AODV as it forwards the overhead packets to all the neighboring nodes. In CBR only the gateway nodes or cluster heads will receive overhead packets. Even in this case as the node density increases the number of gateway nodes also increase and the overhead ratio therefore increases. SBCR chooses only a limited number of gateway nodes and cluster heads based on the number of sectors. The overhead in this case is therefore less when compared to the other two. The number of transmission attempts is higher for SBCR as the survivability (or simulation time) is very high compared to AODV and CBR.

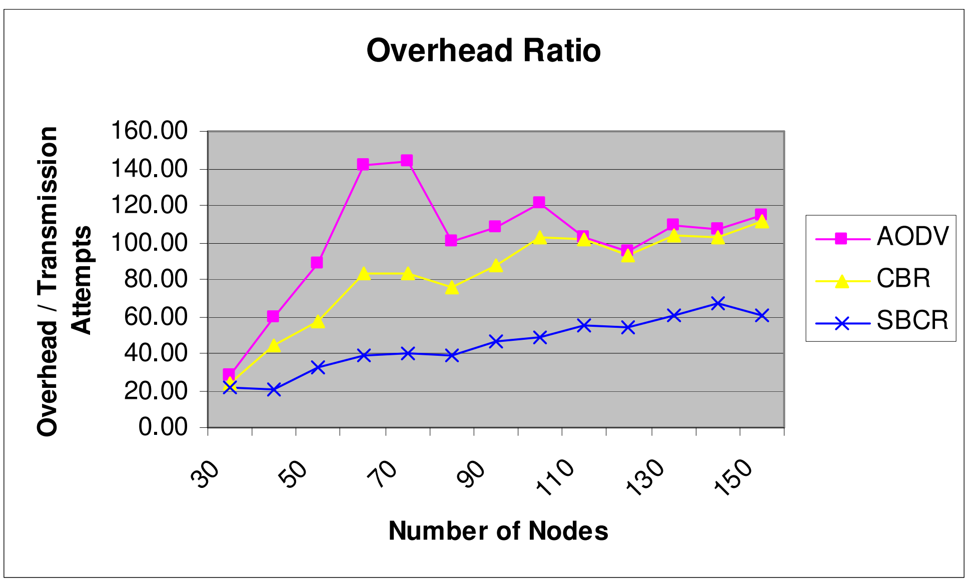

Figure 7.10. Overhead ratios – 5% of nodes dead.

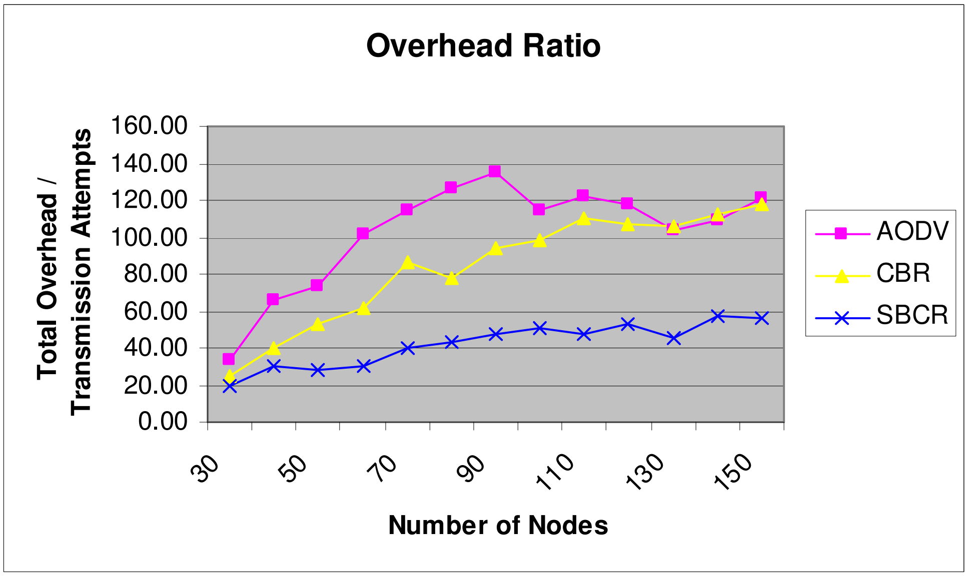

Figure 7.11. Overhead ratios – 10% of nodes dead.

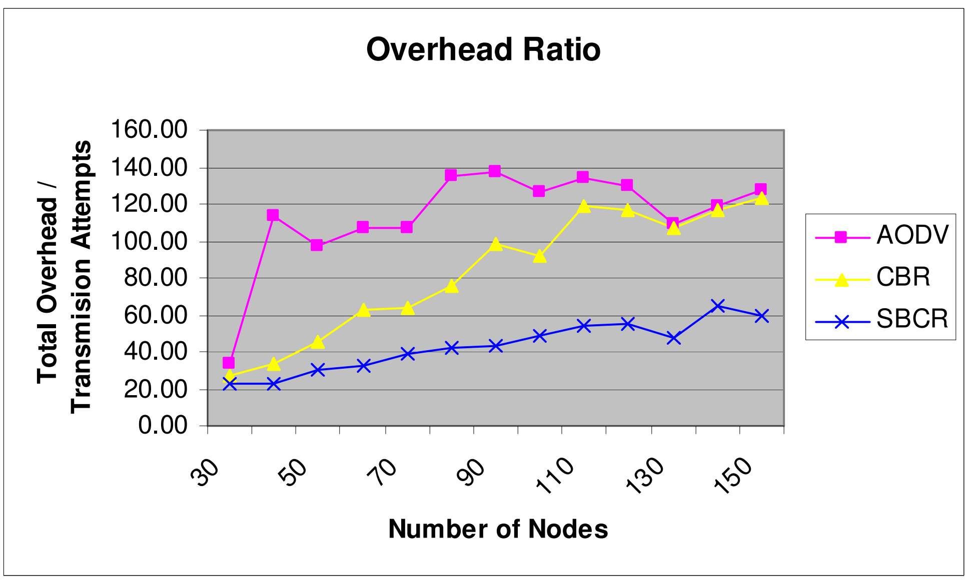

Figure 7.12. Overhead ratios – 25% of nodes dead.

7.1.5 SBCR Paths

This is the average number of routes SBCR provides for data transmission between source and destination. In both AODV and CBR only a single path is provided for data transmission. SBCR uses multiple paths to provide data transfer and figures 7.13, 7.14, and 7.15 show how the paths are affected with increase in node density. The values for these plotted graphs are given in appendix C tables C.13, C.14, and C.15 respectively. From the graphs it is observed that the average number of paths provided increase with the number of nodes in the network. As the node density is high we have multiple options of routes between source and destination. The route discovery process selects reliable routes in a controlled fashion.

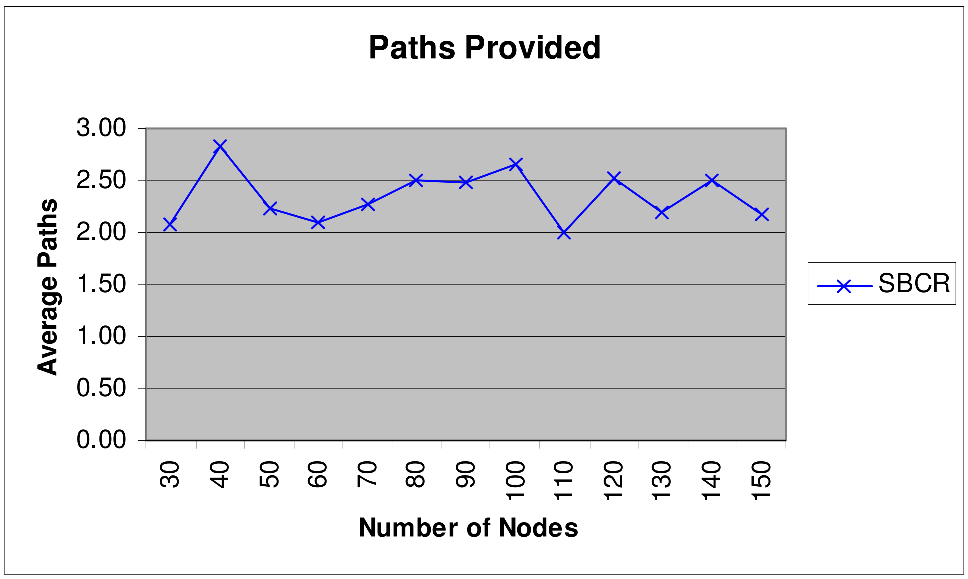

Figure 7.13. SBCR paths – 5% nodes dead.

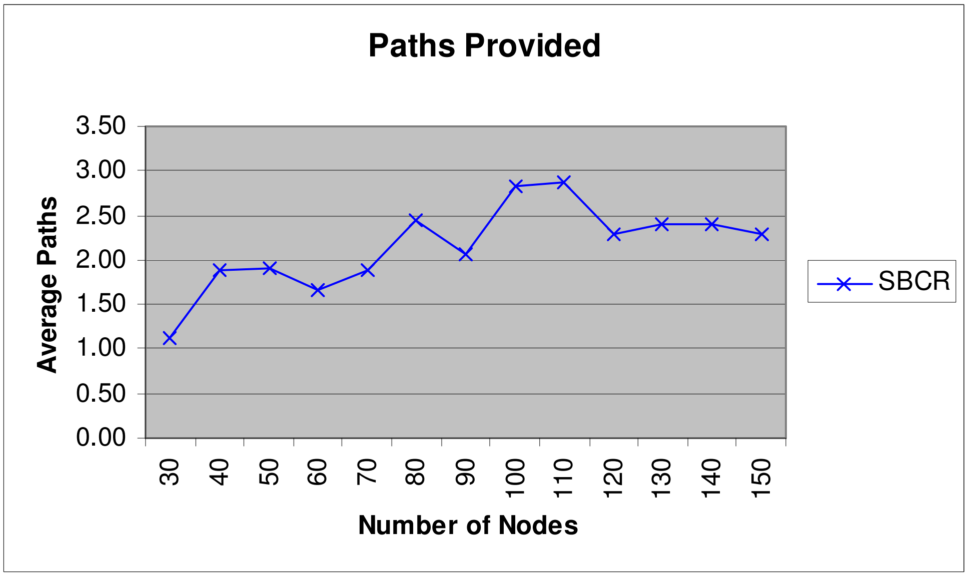

Figure 7.14. SBCR paths – 10% nodes dead.

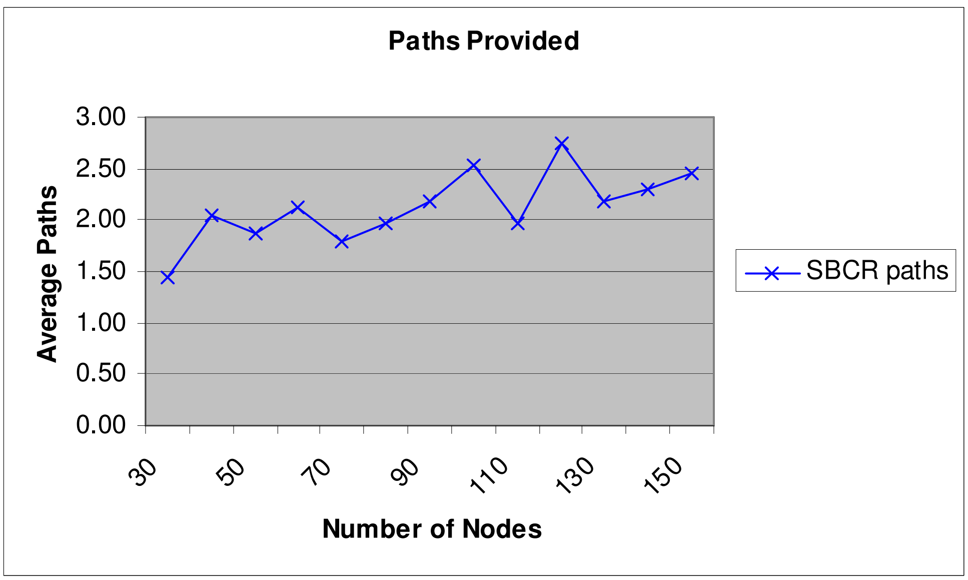

Figure 7.15. SBCR paths – 25% nodes dead.

7.2 Type II

In Type II simulation the simulations were done for a fixed period of time and the results were documented for the three protocols. The results like the number of nodes dead, average node power, message success ratio, overhead ratio, and average number of paths provided are collected after the simulation. The simulations were performed with nodes ranging from 30 to 150 with increments of 10. These simulations with a maximum clock value provide a better understanding of the protocol performance in high density networks. The input values kept fixed for all these simulations are boundary dimensions (1000 by 1000). The maximum security level is set to 2 and the nodes communication radius is restricted to 200. For SBCR the number of sectors is 6 and nodes per sector is 2. The maximum node speed is 3 and the initial power level is 100000 units. The graphs for these simulations are given in the sections below. The number of nodes are given on X axis and results collected are plotted on Y axis for these graphs.

7.2.1 Network Survivability

The number of nodes in the network with power level less than zero after the simulation provides a better understanding of the network survivability. The graph given in figure 7.16 below indicates the protocols performance. The values for these graphs are given in table C.16 appendix C. AODV and CBR have similar low performance results when compared to SBCR. As the node density increases the degree of associativity among nodes is high and so the overhead is more for AODV and CBR. This is not the case in SBCR.

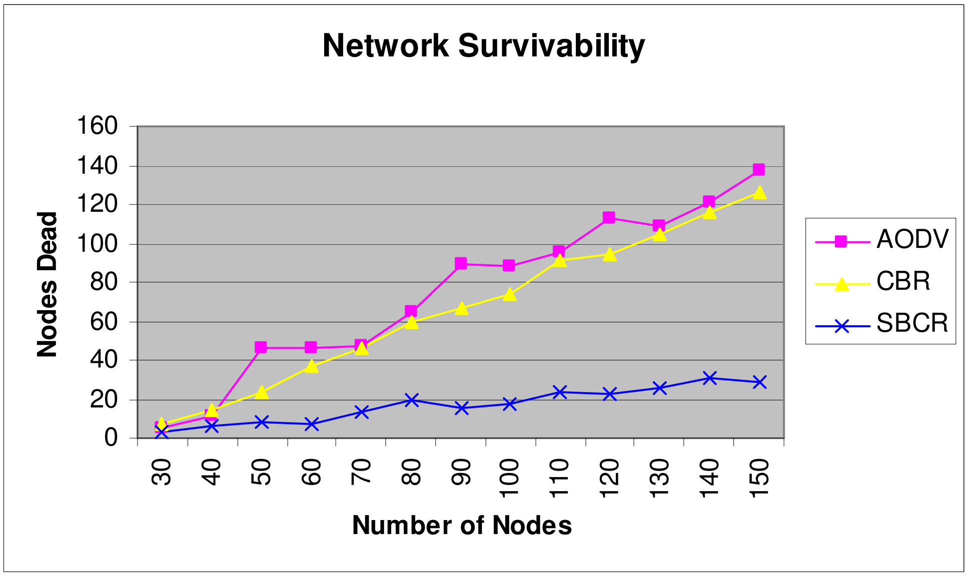

Figure 7.16. Network survivability.

7.2.2 Average Node Power

The average power a node has at the end of the simulation. The graph given in figure 7.17 compares the average node power for the three protocols. The values plotted in this graph are given in table C.17 appendix C. The number of nodes dead is very high in AODV and CBR when compared with SBCR and so the average node power is very less for AODV and CBR. As the node density in the network is increased for each simulation the average power of nodes decreased for all the three protocols. This is because of the increase in the overhead for data transmission.

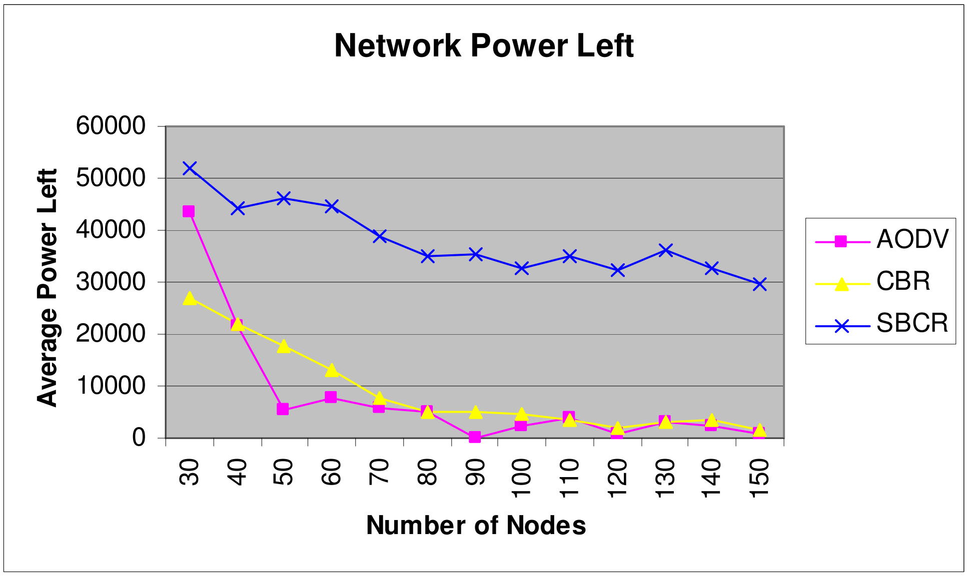

Figure 7.17. Average node power- max time 150.

7.2.3 <u>Message Success Rate</u>

The message success rate results are consistent with those given in Type 1 simulation. The message success rate of AODV and CBR is very low compared to SBCR. This behavior is due to early loss of routes (nodes power zero) during simulation for the AODV and CBR protocols. The source node transmits new messages using the dead routes before it receives an update of the routes. The figure 7.18 plotted from values given in table C.18 appendix C is given below.

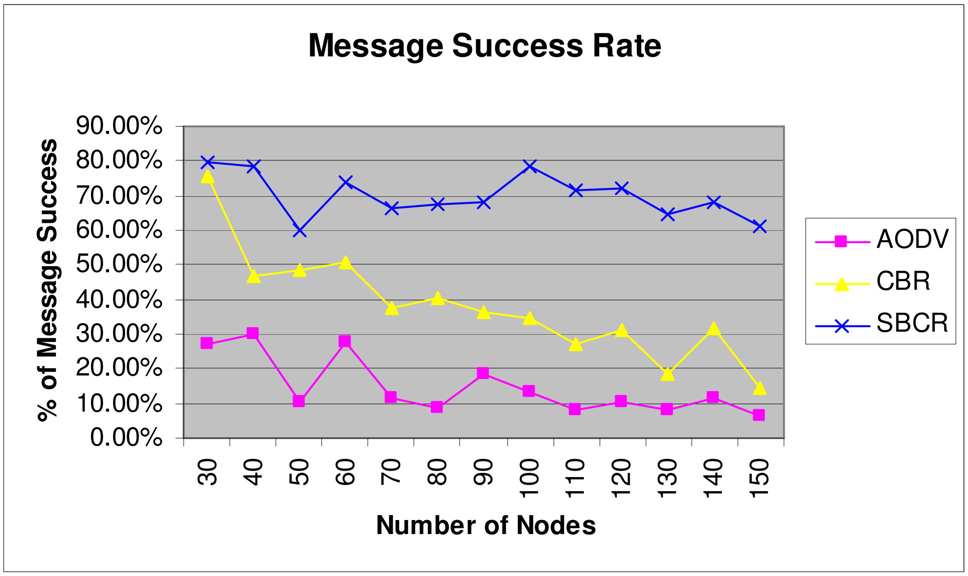

Figure 7.18. Message success ratios – max time 150.

7.2.4 Overhead Ratio

Similar to the results given in Type I simulation, Type II has AODV with high overhead ratio when compared to CBR and SBCR. The SBCR has the lowest overhead ratio among the three protocols. The figure 7.19 has graphs with curves corresponding to AODV, CBR, and SBCR overhead ratios. The values of these graphs are given in table C.19 appendix C. The graphs suggest that as the node density was increased in the network the overhead ratio increased proportionally.

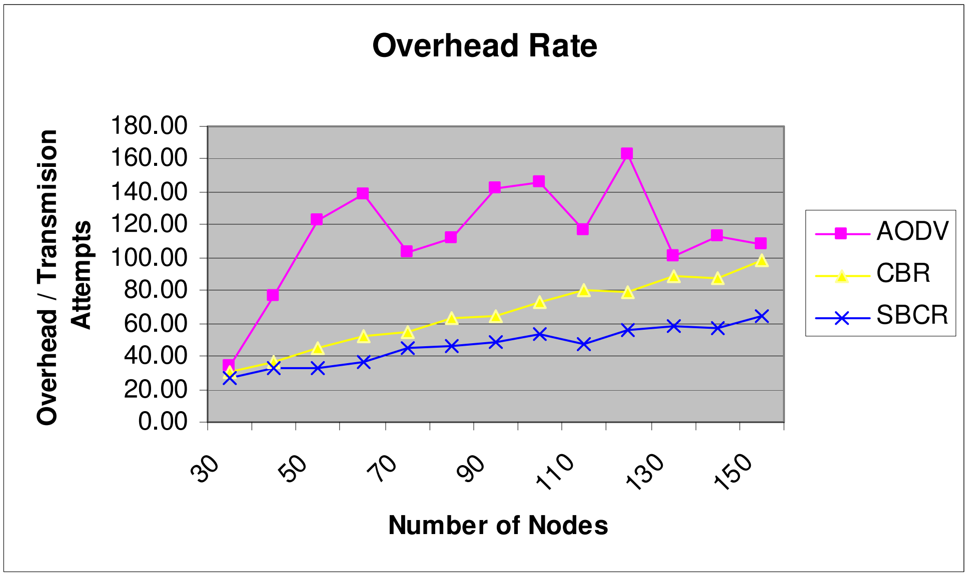

Figure 7.19. Overhead ratios – max time 150.

7.2.5 SBCR Paths

Similar to Type I results, Type II results have an increase in the average number of paths provided, with the increase in node density of the network for every consecutive simulation. The figure 7.20 given below has the average number of paths provided by SBCR on Y axis. The values plotted in this graph are given in table C.20 in appendix C.

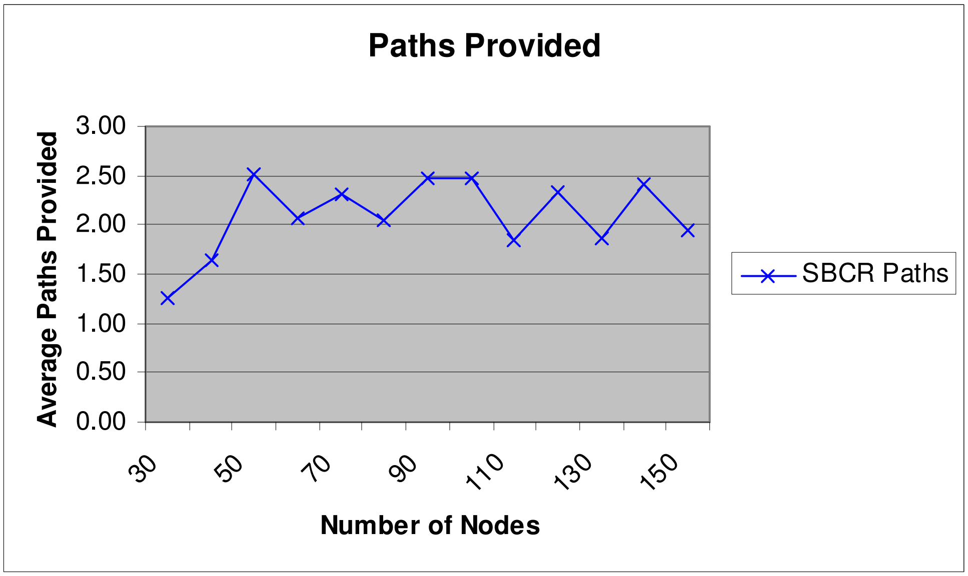

Figure 7.20. SBCR average paths – max time 150.

7.3 Type III

In Type III simulation instead of varying the number of nodes in the network, the number of sectors each node has was varied. Simulations were conducted with varying number of sectors starting from 2 to 12 with increments of 1. These simulations were done specifically for SBCR as it is the only protocol that considers the concept of sectors. These simulations are performed to understand the importance of sectors in SBCR. The influence sectors have on number of nodes dead, average node power, message success ratio, overhead ratio, and average number of paths provided after each simulation was emphasized. The input values kept fixed for all these simulations are the boundary dimensions 1000 by 1000 units. The maximum security level is set to 2 and nodes radius of communication is restricted to 200. The members per sector a cluster head can accept is set to 2. The maximum nodes speed is 3 and the initial power level is 100000 units.

The number of nodes in the network is 100 and the maximum clock time is 150. The graphs for these simulations are given in the sections below and have the number of sectors on X axis along with one of the collected result on Y axis.

7.3.1 Network Survivability

The simulation is similar to Type II except instead of having variable number of nodes it was kept constant as 100 and the number of sectors was changed. The number of sectors has shown affect on the network survivability. This is evident from the figure 7.21 given below, whose values are given in table C.21 appendix C. There is a sharp drop in the number of nodes dead when the number of sectors is changed from 4 to 5. This is because of change in policy for forwarding messages to nodes. When the number of sectors is less than 5 then the messages are forwarded to more than one node, so the overhead is high and also the number of paths provided is less. This causes power drain in many nodes. If the number of sectors is 5 or more then messages are forwarded to only one node. This reduces the overhead and also the number of paths provided is high due to lack of power drain. So having sectors more than 5 had better performance. If the sectors were kept increasing the overhead also increases and so we have slight increase in number of nodes dead during simulation.

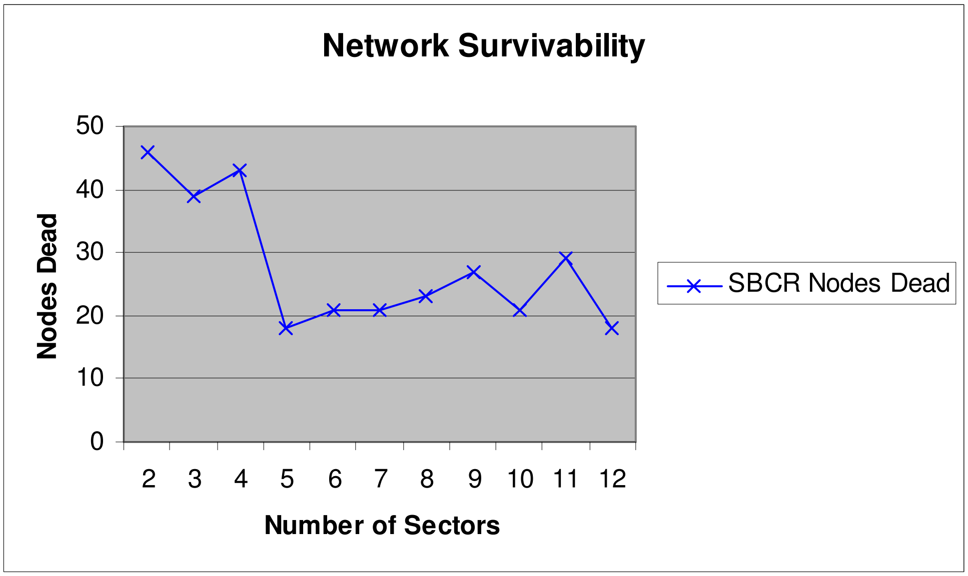

Figure 7.21. Network survivability - number of sectors.

7.3.2 Average Node Power

The graph in figure 7.22 has the average node power on Y axis. The values plotted on the graph are given in table C.22 appendix C. From the graph we have the number of nodes in the network dead is high when the number of sectors is less than 5 the average power is also low. There is a steep increase in the average power level when the number of sectors is above 4. The number of paths provided depends on the number of sectors. If we have multiple paths then the data transmission cost is equally shared by many nodes. Also the overhead of forwarding messages is less as explained above.

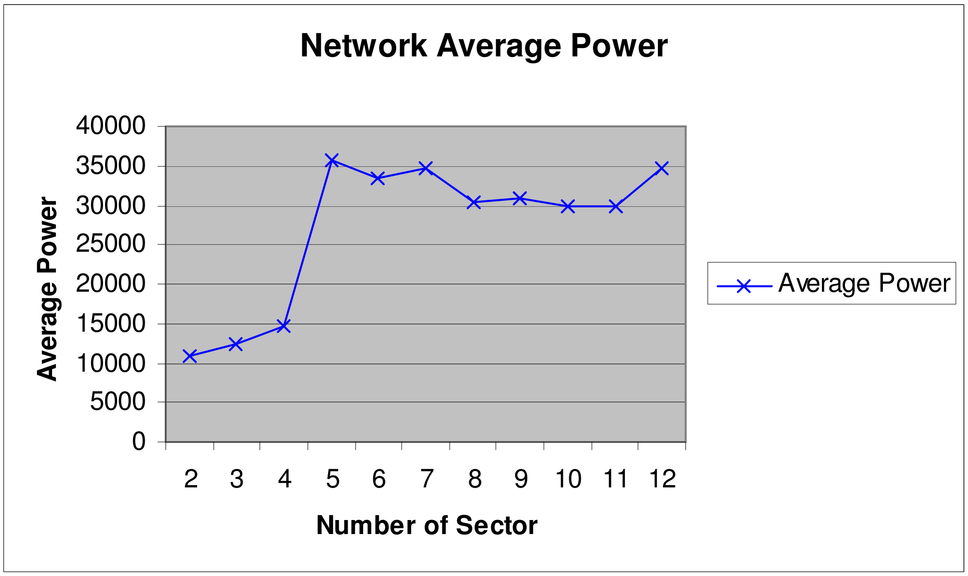

Figure 7.22. Average node power – number of sectors.

7.3.3 Message Success Rate

The graph in figure 7.23 suggests that the message success rate is increased with the increase in number of sectors. The number of paths provided having sectors more than 4 is high when compared with less than 5 sectors. So we can conclude that larger the number of paths provided the better the message success rate we obtain. The values plotted in the graph are given in table C.23 appendix C.

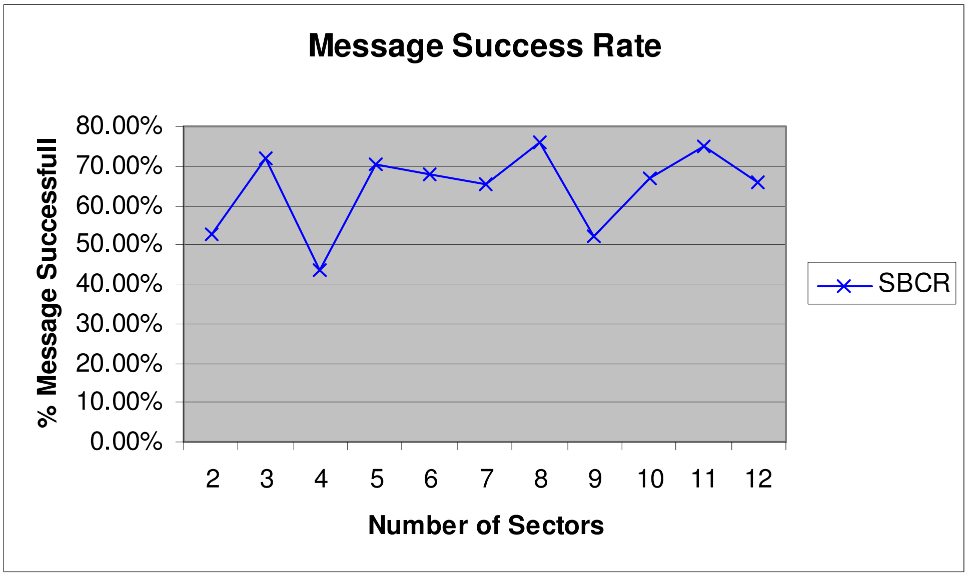

Figure 7.23. Message success rate – number of sectors.

7.3.4 Overhead Ratio

The overhead rate increases with the increase in number of sectors. When the number of sectors is less than 5 then the overhead rate is very high. This is because of the SBCR policy of forwarding message to more than one node. When the number of sectors is greater than 4 then the message is forwarded to only one node in each sector. So the message traverses in all the directions covering very less number of nodes. Further increase in number of sectors increases the overall number of nodes a message is forwarded to. This is the reason for the increase in overhead ratio with the increase in number of sectors. The graph in figure 7.24 has the overhead ratio on Y axis and the number of sectors on X axis. The values plotted in the graph are given in table C.24 of appendix C.

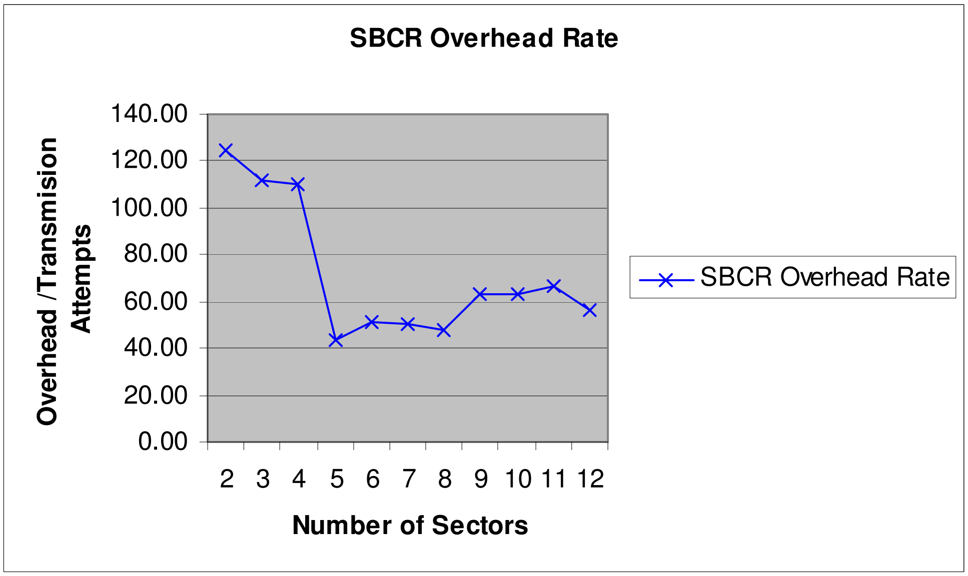

Figure 7.24. Overhead rate – number of sectors.

7.3.5 Average Paths

The average number of paths provided by SBCR during data transmission for the entire simulation affects the message success rate, number of nodes dead, and average node power. The number of sectors used in a simulation affects the number of paths provided for data transmission. The maximum number of paths provided depends on the number of sectors and security level of a node. The graph in figure 7.25 indicates an increase in average paths with increase in number of sectors. The values used for plotting this graph are given in table C.25 of appendix C.

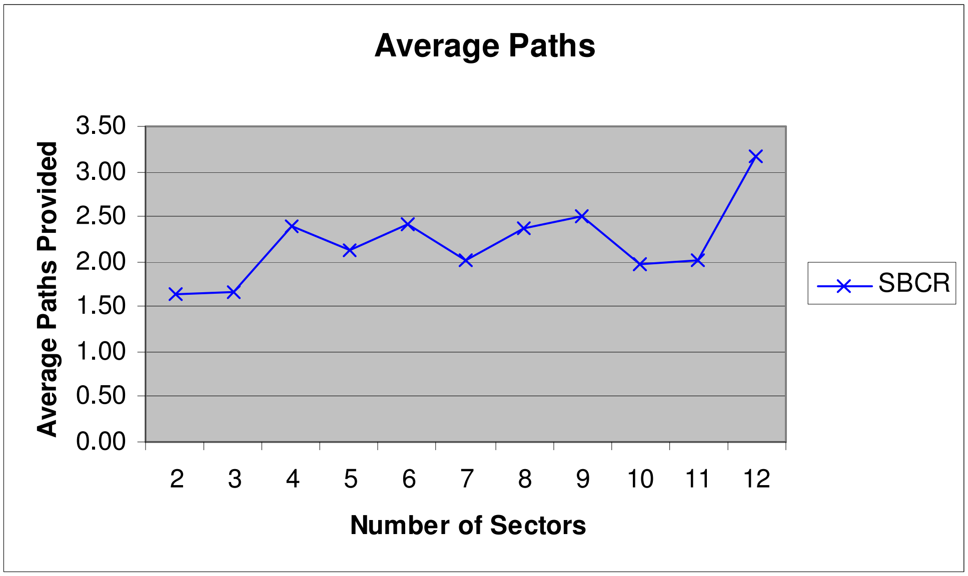

Figure 7.25. Average paths – number of sectors.

7.4 Complexity

The complexity of SBCR can be given individually for its subcomponents. The cluster head decision process that is part of sector based clustering has a complexity of O (N^2); Where N is the number of sectors. The member inclusion has a complexity of O (K) where K is the number of nodes in a sector. The route discovery process present in the sector based routing has a complexity of O (d * M) where M is the total number of nodes present in the sectors that satisfy the range conditions and d is the number of hops between source and destination. The rest of the processes do not have significant complexity associated with them.

CHAPTER VIII

SUMMARY AND FUTURE WORK

This chapter includes the summary of this work and also gives an insight into the future related to this area.

8.1 Summary

The existing clustering and routing protocols have routing of information in ad hoc networks as their primary goal. In recent years protocols like secure routing and power aware routing were developed to address security and power issues of ad hoc networks. All these protocols were only able to address one issue, that is, either power or security. Developing a new approach for clustering that considers the security and power issues of ad hoc networks is the primary objective of this work. The protocol should be able to provide an infrastructure using which any stochastic routing algorithm will be capable of providing additional security to data transferred in ad hoc networks. Moreover, a routing protocol that utilizes the proposed clustering protocol should provide the additional security while increasing the network survivability by conserving power. The present routing algorithms have a tendency to route data using few routes which causes power drain in that path and ultimately leads to splitting the network. Isolated nodes or groups of nodes will be formed because of splitting, which is not healthy for communication among nodes present in a network. Avoiding power drain is therefore another objective of the routing algorithm. The proposed sector based clustering and routing protocols accomplish the above mentioned objectives. The simulations conducted on these protocols confirm the efficiency and advantages of these algorithms and suggest that sector based clustering and routing as potential next generation mobile ad hoc network protocols.

8.2 Future Work

The sector based clustering and routing protocols can be further improved by enabling some Quality of Service features in them. The multi-path routing algorithm can be further improved by taking into consideration the QoS aspects of mobile ad hoc networks. Artificial intelligence concepts can be utilized in the cluster head decision process which may further reduce the overhead of cluster head election.

REFERENCES

[Amis et al. 00] Alan D. Amis, Ravi Prakash, Thai H.P. Vuong, and Dung T. Huynh, "Max-Min D-Cluster Formation in Wireless Ad Hoc Networks", *Proceedings of the Nineteenth Annual Joint Conference of the IEEE Computer and Communication Societies*, pp. 32-41, Tel-Aviv, Israel, March 2000.

[Alzoubi et al. 02] Khaled M. Alzoubi, Peng-Jun Wan, and Ophir Frieder, "New Distributed Algorithm for Connected Dominating Set in Wireless Ad Hoc Networks", *Proceedings of the 35th Hawaii International Conference on System Sciences*, pp. 3849- 3855, Big Island Hawaii, January 2002.

[Banerjee and Khuller 01] Suman Banerjee and Samir Khuller, "A Clustering Scheme for Hierarchical Control in Multi-hop Wireless Networks", *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, pp. 1028–1037, Anchorage, Alaska, April 2001.

[Barrett et al. 03] Christopher L. Barrett, Stepahn J. Eidenbenz, Lukas Kroc, Madhav Marathe, and James P. Smith, "Parametric Probabilistic Sensor Network Routing", *Proceeding of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*, pp. 122-131, San Diego, CA, September 2003.

[Bohacek et al. 02] Stephan Bohacek, Joao P. Hesanha, Katia Obraczka, Junsoo lee, and Chansook Lim, "Enhancing Security via Stochastic Routing", *Proceedings of the Eleventh International Conference on Computer Communications and Networks*, pp. 58-62, Miami, Florida, October 2002.

[Camp et al. 02] Tracy Camp, Jeff Boleng, Vanessa Davies, "A survey of mobility models for ad hoc network research", Dept. of Math and Computer Sciences, Colorado School of Mines, Golden, CO., September 2002. *http://toilers.mines.edu/papers/pdf/Models.pdf*

[Chatterjee et al. 02] Mainak Chatterjee, Sajal K. Das, and Damla Turgut, "WCA: A Weighted Clustering Algorithm for Mobile Ad Hoc Networks", *Cluster Computing*, Vol. 5, No. 2, pp. 193-204, April 2002.

[Chen et al. 02] Geng Chen, Fabian Garcia Nocetti, Julio Solano Gonzalez, and Ivan Stojmenovic, "Connectivity Based k-hop Clustering in Wireless Networks", *Proceedings of the 35th Hawaii International Conference on System Sciences*, pp. 2450-2459, Big Island, Hawaii, January 2002.

[Chen and Liestman 02] Yuanzhu Peter Chen and Arthur L. Liestman, "Approximating Minimum Size Weakly-Connected Dominating Sets for Clustering Mobile Ad Hoc Networks", *Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing*, pp. 165-172, Lausanne, Switzerland, June 2002.

[Corson 02] Scott Corson, "An Overview of Mobile Ad Hoc Networking", *http://inet2002.org/CD-ROM/lu65rw2n/papers/t13-a.pdf,* creation date: unknown, last update: June 2002, access date: March 20th 2004.

[Deng et al. 02] Hongmei Deng, Wei Li, and D. P. Agrawal, "Routing Security in Wireless Ad hoc networks", *Communications Magazine*, IEEE, Vol. 10, No. 2, pp. 70-75, October 2002.

[Gerla and Tsai 95] Mario Gerla and Jack Tzu-Chieh Tsai, "Multicluster, mobile, multimedia radio network", *Wireless Networks*, Vol. 1, No. 3, pp. 255-265, March 1995.

[Kim et al. 98] Dongkyun Kim, Seokjae Ha, and Yanghee Choi, "K-hop Cluster-based Dynamic Source Routing in Wireless Ad-Hoc Packet Radio Network", *Proceedings of the 48th IEEE conference on Vehicular Technology*, pp. 224-228, Ottawa, Ont. , Canada, May 1998.

[Krishna et al. 97] P. Krishna, N.H. Vaidya, M. Chatterjee, and D.K. Pradhan, "A Cluster-based Approach for Routing in Dynamic Networks", *ACM SIGCOMM Computer Communication Review*, Vol. 27, No. 2, pp. 49-64, April 1997.

[Papadimitratos and Haas 02] Panagiotis Papadimitratos and Zygmunt J. Hass, "Secure Routing for Mobile Ad Hoc Networks", *Proceedings of the SCS Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2002)*, pp. San Antonio, TX, January 2002.

[Perkins 01] Charles E. Perkins, "Ad hoc Networking", Addison-Wesley, Upper Saddle River, NJ, December 2001.

[Perkins and Royer 99] Charles E. Perkins and Elizabeth M. Royer, "Ad-hoc On-Demand Distance Vector Routing" *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications,* pp. 90, Washington, DC, USA, 1999.

[RoamAbout 96] RoamAbout 2.4 GHz frequency hopping wireless LAN adapters, Digital Equipment Cooperation, EC-F5700-42, 1996.

[Royer and Toh 99] Elizabeth M. Royer and Chai-Keong Toh, "A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks", *Personal Communications*, IEEE, Vol. 6, No. 2, pp. 46-55, April 1999.

[Sucec and Marsic 02] John Sucec and Ivan Marsic, "Clustering Overhead for Hierarchical Routing in Mobile Ad hoc Networks", *Proceedings of the Twenty-*

First Annual Joint Conference of the IEEE Computer and Communications Societies, pp. 1698 -1706, New York, NY, June 2002.

[Tsudaka et al. 01] Kentaro Tsudaka, Masanobu Kawahara, Akira Matsumoto, and Hiromi Okada, "Power Control Routing for Multi Hop Wireless Ad-hoc Network", *Proceeding of the Global Telecommunications Conference*, pp. 2819-2824, San Antonio, TX, November 2001.

[Wireless 03] Wireless Ad Hoc Networks*, http://w3.antd.nist.gov/wctg/manet/*, creation date: January 15th 2003, access date: March 20th 2004.

APPENDICES

APPENDIX A

GLOSSARY

ABR	Associativity Based Routing
Adjacency Value	Indicates the relative movement of a node with respect to the other node
AODV	Ad Hoc On-Demand Distance Vector Routing
CBR	Cluster Based Routing.
CDMA	Code Division Multiple Access is a standard in cellular communication
CGSR	Clusterhead Gateway Switch Routing
Cluster	A group of nodes present in the same geographical region
Clusterhead	A node that acts as a center of communication for a group of nodes
DARPA	Defense Advance Research Project Agency a United States government organization that promotes research in latest technologies
DSDV	Destination-Sequenced Distance-Vector Routing
DSR	Dynamic Source Routing
GSM	Global System for Mobile communication is a standard in cellular communication

MANET	Mobile Ad Hoc Networks are a group of mobile nodes that can communicate without the support of basic infrastructure
NDV	Node Distribution Value indicates how evenly the nodes are distributed around a node
Power Level	The power remaining in the battery of a node
SBCR	Sector Based Clustering and Routing
Sector	The division of area surrounding a node.
Security Level	The level of security that should be provided to a node.
SSR	Signal Stability Routing
TDMA	Time Division Multiple Access is a standard in cellular communication
TORA	Temporally Ordered Routing Algorithm
WRP	Wireless Routing Protocol

APPENDIX B

PROGRAM OUTPUT

The output of the simulation after a single run is given below.

```
Ad hoc On-Demand Distance Vector Routing Protocol Simulation
Number of Nodes: 100
Simulation Time: 33
Number of Nodes Dead: 25
Average Power Left of a Node: 17968
Number of Message Sent: 1207
Number of Message reached Destination: 304
Number of Message dropped: 0
Total Overhead: 12639
Total Transmission attempts: 100
Total Messages lost in the Network: 2351
% of Messages transmitted successfully: 25.1864

 Cluster Based Routing Protocol Simulation
Number of Nodes: 100
Simulation Time: 37
Number of Nodes Dead: 26
Average Power Left of a Node: 19547
Number of Message Sent: 1792
Number of Message reached Destination: 889
Number of Message dropped: 83
Total Overhead: 10062
Total Transmission attempts: 109
Total Messages lost in Network: 1713
% of Messages transmitted successfully: 49.6094

 Sector Based Clustering & Routing Protocol Simulation
Number of Nodes: 100
Simulation Time: 177
Number of Nodes Dead: 25
Average Power Left of a Node: 30816
Number of Message Sent: 7977
Number of Message reached Destination: 5101
Number of Message dropped: 443
Total Overhead: 11354
Total Transmission attempts: 234
% of Messages transmitted successfully: 63.9463
Total Number of paths: 109
Total Transmissions: 43
Total Message lost in Network: 1334
Average number of paths provided: 2.53488
```

APPENDIX C

RESULTS TABLES

The results gathered for the simulation are given in tables below:

Type I:

Nodes	Simulation Time		
	AODV	CBR	SBCR
30	135	71	145
40	77	45	120
50	54	37	136
60	47	34	98
70	35	34	126
80	36	33	107
90	32	30	88
100	26	27	93
110	21	20	103
120	17	16	81
130	19	18	94
140	16	14	79
150	15	13	85

Table C.1. Network survival – 5% of nodes dead.

Nodes	Simulation Time		
	AODV	CBR	SBCR
30	100	91	158
40	75	44	101
50	64	42	110
60	63	54	121
70	45	36	135
80	34	36	105
90	33	34	126
100	29	28	112
110	23	22	129
120	21	20	105
130	16	15	119
140	15	14	106
150	16	14	106

Table C.2. Network survival – 10% of nodes dead.

Nodes	Simulation Time		
	AODV	CBR	SBCR
30	219	192	230
40	128	131	208
50	103	67	202
60	82	53	211
70	62	52	177
80	49	49	185
90	46	43	169
100	33	37	177
110	30	32	169
120	26	26	177
130	19	18	175
140	20	19	151
150	19	18	164

Table C.3. Network survival – 25% of nodes dead.

Nodes	Average Power Left		
	AODV	CBR	SBCR
30	57879	62479	55602
40	51254	60309	56814
50	45717	50809	51997
60	36469	48221	62951
70	24118	39790	50818
80	39778	40182	56869
90	34138	34614	57195
100	22277	23807	56503
110	31291	33045	50884
120	35760	34779	62315
130	25821	25762	49628
140	23068	25233	54254
150	18646	19994	61017

Table C.4. Average power left -5% of nodes dead.

Nodes	Average Power Left		
	AODV	CBR	SBCR
30	60390	54755	58535
40	47097	47324	56476
50	46274	47593	51999
60	35320	37553	55616
70	30436	39128	50313
80	28874	31987	50801
90	24860	27237	52050
100	23459	24418	47482
110	23734	27934	45103
120	20208	22363	49032
130	24339	23684	47190
140	23996	22838	49171
150	17335	20506	48296

Table C.5. Average power left – 10% of nodes dead.

Nodes	Average Power Left		
	AODV	CBR	SBCR
30	30620	28822	34210
40	26244	35595	32862
50	19414	32572	32303
60	24779	34594	25986
70	26117	25977	29220
80	21857	23588	28974
90	22675	23414	31155
100	17968	19547	30816
110	19402	18734	32509
120	15135	16653	27051
130	18797	19055	27822
140	15847	17377	29999
150	12349	12980	27226

Table C.6. Average power left – 25% of nodes dead.

Nodes	% of Message Trans Successfully		
	AODV	CBR	SBCR
30	39.70	89.32	75.83
40	43.98	78.83	93.15
50	30.21	64.74	91.74
60	25.23	59.45	89.73
70	20.76	63.76	93.85
80	28.58	70.66	92.59
90	17.19	54.11	86.33
100	25.95	51.21	90.58
110	23.42	48.30	88.13
120	30.08	44.80	79.02
130	37.28	52.62	85.95
140	31.60	39.33	86.29
150	43.35	46.85	88.60

Table C.7. Message success ratio – 5% of nodes dead.

Nodes	% of Message Trans Successfully		
	AODV	CBR	SBCR
30	19.07	49.59	67.57
40	19.93	75.12	91.21
50	14.39	71.45	88.72
60	24.98	56.25	86.66
70	21.68	45.22	84.68
80	26.30	50.75	85.99
90	19.64	54.73	83.11
100	23.96	50.79	82.35
110	27.86	49.05	81.26
120	21.26	40.23	77.04
130	26.36	44.29	77.88
140	28.36	41.40	74.39
150	30.05	43.48	76.30

Table C.8. Message success ratio – 10% of nodes dead.

Nodes	% of Message Trans Successfully		
	AODV	CBR	SBCR
30	32.79%	85.20%	81.89%
40	42.85%	81.78%	80.35%
50	27.08%	53.49%	75.97%
60	26.30%	55.49%	66.05%
70	18.47%	52.24%	59.62%
80	14.44%	47.92%	68.91%
90	18.03%	51.57%	69.17%
100	25.19%	49.61%	63.95%
110	20.35%	37.98%	68.67%
120	23.04%	42.86%	59.88%
130	28.39%	40.38%	58.21%
140	35.19%	43.42%	73.44%
150	20.54%	35.02%	66.60%

Table C.9. Message success ratio – 25% of nodes dead.

Nodes	Overhead			Transmission Attempts			Overhead/Transmission attempts		
	AODV	CBR	SBCR	AODV	CBR	SBCR	AODV	CBR	SBCR
30	2108	1163	1626	76	49	74	27.74	23.73	21.97
40	4049	1756	1737	68	40	85	59.54	43.90	20.44
50	5784	2856	3725	65	50	114	88.98	57.12	32.68
60	8766	4981	4219	62	60	108	141.39	83.02	39.06
70	10051	5834	6436	70	70	161	143.59	83.34	39.98
80	8028	6038	5641	80	80	144	100.35	75.48	39.17
90	9629	7908	6455	89	90	140	108.19	87.87	46.11
100	12060	10259	8156	100	100	169	120.60	102.59	48.26
110	10982	10924	10436	107	107	190	102.64	102.09	54.93
120	10945	10991	10514	115	118	193	95.17	93.14	54.48
130	13584	13206	12362	124	127	203	109.55	103.98	60.90
140	13898	13290	14307	130	130	215	106.91	102.23	66.54
150	14754	14384	14235	129	129	235	114.37	111.50	60.57

Table C.10. Overhead ratio – 5% of nodes dead.

Nodes	Overhead			Transmission Attempts			Overhead/Transmission attempts		
	AODV	CBR	SBCR	AODV	CBR	SBCR	AODV	CBR	SBCR
30	1766	1464	1548	53	59	78	33.32	24.81	19.85
40	3407	1640	2005	52	41	66	65.52	40.00	30.38
50	4546	2659	2433	62	50	85	73.32	53.18	28.62
60	8867	5674	3828	87	92	127	101.92	61.67	30.14
70	8030	6336	6377	70	73	159	114.71	86.79	40.11
80	10160	7053	5740	80	90	134	127.00	78.37	42.84
90	12159	8635	8949	90	92	190	135.10	93.86	47.10
100	11508	9914	9861	100	101	194	115.08	98.16	50.83
110	13447	11969	10616	110	109	222	122.25	109.81	47.82
120	13700	12643	11023	116	118	207	118.10	107.14	53.25
130	12628	12671	11001	122	120	244	103.51	105.59	45.09
140	13984	13855	13289	128	123	231	109.25	112.64	57.53
150	16371	15503	14476	135	131	260	121.27	118.34	55.68

Table C.11. Overhead ratio – 10% of nodes dead.

Nodes	Overhead			Transmission Attempts			Overhead/Transmission attempts		
	AODV	CBR	SBCR	AODV	CBR	SBCR	AODV	CBR	SBCR
30	2814	2498	2414	85	94	106	33.11	26.57	22.77
40	9883	3086	2569	87	91	114	113.60	33.91	22.54
50	9092	3719	4174	93	82	136	97.76	45.35	30.69
60	8577	4176	5034	80	67	155	107.21	62.33	32.48
70	9441	6176	5893	88	97	150	107.28	63.67	39.29
80	11342	7233	8192	84	96	192	135.02	75.34	42.67
90	12327	9422	8939	90	96	206	136.97	98.15	43.39
100	12639	10062	11354	100	109	234	126.39	92.31	48.52
110	14559	12964	12635	109	109	235	133.57	118.94	53.77
120	15483	13751	16215	119	118	292	130.11	116.53	55.53
130	13582	13312	13074	124	124	278	109.53	107.35	47.03
140	16150	15394	19405	136	132	301	118.75	116.62	64.47
150	17673	17008	20283	138	138	339	128.07	123.25	59.83

Table C.12. Overhead ratio – 25% of nodes dead.

Nodes	SBCR-Paths
30	2.08
40	2.83
50	2.24
60	2.10
70	2.27
80	2.50
90	2.49
100	2.65
110	2.00
120	2.53
130	2.20
140	2.49
150	2.17

Table C.13. SBCR paths – 5% nodes dead.

Nodes	SBCR-paths
30	1.13
40	1.88
50	1.90
60	1.65
70	1.89
80	2.44
90	2.06
100	2.84
110	2.88
120	2.28
130	2.40
140	2.41
150	2.28

Table C.14. SBCR paths – 10% nodes dead.

Nodes	SBCR-paths
30	1.44
40	2.05
50	1.88
60	2.13
70	1.80
80	1.97
90	2.17
100	2.53
110	1.96
120	2.75
130	2.18
140	2.29
150	2.46

Table C.15. SBCR paths – 25% nodes dead.

Type II:

Nodes	Nodes Dead		
	AODV	CBR	SBCR
30	5	7	3
40	11	14	6
50	46	24	8
60	46	37	7
70	47	46	13
80	65	60	19
90	89	67	15
100	88	74	17
110	95	91	24
120	113	94	23
130	109	105	26
140	121	116	31
150	137	126	29

Table C.16. Network survivability.

Nodes	Average Power Left		
	AODV	CBR	SBCR
30	43421	26995	51874
40	21398	21739	44159
50	5320	17654	45964
60	7850	12916	44704
70	5721	7660	38853
80	5081	4860	35171
90	38	4999	35231
100	2141	4442	32560
110	3754	3503	35070
120	597	2049	32152
130	3128	2909	36097
140	2287	3577	32686
150	728	1639	29707

Table C.17. Average node power-max time 150.

Nodes	% Message of Trans Successfully		
	AODV	CBR	SBCR
30	27.36%	75.62%	79.76%
40	30.04%	46.91%	78.66%
50	10.44%	48.46%	59.84%
60	27.43%	50.67%	74.13%
70	11.49%	37.60%	66.25%
80	8.55%	40.27%	67.35%
90	18.25%	36.28%	67.79%
100	13.51%	34.39%	78.53%
110	8.18%	26.92%	71.59%
120	10.67%	31.16%	72.08%
130	8.07%	18.31%	64.87%
140	11.41%	31.91%	67.95%
150	6.12%	14.56%	61.28%

Table C.18. Message success ratio- max time 150.

Nodes	Overhead			Transmission Attempts			Overhead/Trans attempts		
	AODV	CBR	SBCR	AODV	CBR	SBCR	AODV	CBR	SBCR
30	2281	2441	2049	68	79	76	33.54	30.90	26.96
40	6704	3459	3308	87	95	102	77.06	36.41	32.43
50	7492	5828	3357	61	128	103	122.82	45.53	32.59
60	15077	7444	4819	109	141	132	138.32	52.79	36.51
70	11429	9365	6645	110	173	149	103.90	54.13	44.60
80	12588	10642	7841	113	168	169	111.40	63.35	46.40
90	15973	12245	9012	112	189	185	142.62	64.79	48.71
100	19771	14330	11836	136	197	222	145.38	72.74	53.32
110	17890	15715	10600	153	195	222	116.93	80.59	47.75
120	23309	18094	14728	143	228	261	163.00	79.36	56.43
130	19532	18740	16370	193	211	281	101.20	88.82	58.26
140	19693	19120	16205	175	218	281	112.53	87.71	57.67
150	20767	20531	20494	191	209	319	108.73	98.23	64.24

Table C.19. Overhead ratios – max time 150.

Nodes	SBCR-paths
30	1.25
40	1.65
50	2.52
60	2.07
70	2.31
80	2.05
90	2.47
100	2.48
110	1.85
120	2.33
130	1.87
140	2.41
150	1.95

Table C.20. SBCR average paths provided – max time 150.

Type III

Sectors	Nodes Dead
2	46
3	39
4	43
5	18
6	21
7	21
8	23
9	27
10	21
11	29
12	18

Table C.21. Network survivability-number of sectors.

Sectors	Average Power
2	10983
3	12417
4	14750
5	35807
6	33346
7	34758
8	30443
9	30986
10	29985
11	29825
12	34602

Table C.22. Average node power-number of sectors.

Sectors	% of Message Successful
2	52.52%
3	71.95%
4	43.52%
5	70.26%
6	68.03%
7	65.30%
8	75.71%
9	52.21%
10	67.03%
11	75.13%
12	66.02%

Table C.23. Message success rate – number of sectors.

Sectors	Overhead	Trans Attempt	Overhead/TA
2	27864	224	124.39
3	28448	255	111.56
4	20614	187	110.24
5	8505	197	43.17
6	10146	198	51.24
7	10922	218	50.10
8	9866	207	47.66
9	12130	193	62.85
10	13217	210	62.94
11	14191	214	66.31
12	12444	222	56.05

Table C.24. Overhead ratio – number of sectors.

Sectors	Average Paths
2	1.63
3	1.67
4	2.38
5	2.12
6	2.41
7	2.02
8	2.38
9	2.50
10	1.98
11	2.02
12	3.17

Table C.25. Average paths – number of sectors.

APPENDIX D

PROGRAM LISTING

The simulation program coded in C++ is given as follows.

```
/*
        The Objective of this simulator is to provide a fair
        environment for all the three protocols, Ad hoc On-Demand
        Distance Vector Routing, Cluster Based Routing, and
        Sector Based Clustering & Routing. It simulates all the
        three protocols for the same randomly selected network
        node setup. It has the Options of selecting three different
        types of simulation. Type-I we have the simulation carried
        until certain number of nodes in the network run out of
        power. In Type-II we have simulation carried up to certain
        clock value, which is set priory. Type-III only Sector/
*************************************************************************
Student Name:       SUDHEER KRISHNA CHIMBLI VENKATA

Thesis Title:       SECTOR BASED CLUSTERING & ROUTING

Adviser:    JOHNSON P THOMAS
*************************************************************************
*/

        Based Clustering & Routing is simulated with varying number
        of sector values. Generally Type I and Type II are done with
        varying number of nodes in the network.
        Duration:
        Design: 10 Days.
        Design & Implementation: 20 Days.
        Testing: 10 Days.
        Results & Documentation: 10Days.
*/

/*
 File Name: Node.h
 This is the basic node header file.
 The nodes belonging to all the three protocols have these
 data types and functions associated with them.
*/
# pragma once
# include <iostream>
# include <list>
# include <vector>
# include <math.h>

using namespace std;

// Declaration of the Node class.

class CNode
{
public:
        long iPowerlevel; //Power remaining in a node.
```

```
        int iSecuritylevel; //Security desired by the node.
        int iSpeed; //The speed with which it is travelling.
        int iCurrloc_X; //Current location X co-ordinate.
        int iCurrloc_Y; //Current location Y co-ordiante.
        int iFutloc_X; //Nodes future location X co-ordinate.
        int iFutloc_Y; //Nodes Future location Y co-ordinate.
        int iBoundary_X; //Plain Boundary X co-ordinate limit.
        int iBoundary_Y; //Plain Boundary Y co-ordinate limit.
        int iMaxspeed; //Max speed with which a node can travel.
        bool bMobility; //Flag indicating if the node is mobile.
        long iBeacontime; //The beacon interval.
        long iLastbeacon; //The last beacon sent time stamp.
        int iNodeid;// Node unique id in the network.
        long iPausetime; //The random pause time set after reaching destination.
        long lMessage_succes;//Track of number of successful messages reached.
        long lOverhead; //Number of message overhead recevied.
        long lMessage_drop;//Messages droped.
        long lMessage_sent; //Messages sent.
        long lTotaltrans; //Number of transmissions.
        bool bCounted; //Flag
        //Function declaration.
        CNode(void);
        CNode(const CNode &x1);
        CNode& operator=(CNode x1);
        CNode(int Cx, int Cy, int Nid, int speed, int Bod_x, int Bod_y,long power, long Beacon);
        void Set_future(void);
        void Move_node(void);
        bool Send_beacon(long iTime);

};

/*
        File Name: Node.cpp
   This file has the implementations of functions present in
        CNode class declared in the Node.h file.
*/

# include "Node.h"

// Default Constructor of the CNode class. Sets all the
// initial values.

CNode::CNode(void)
{
 iPowerlevel =0;
 iSecuritylevel=0;
 iSpeed =0;
 iCurrloc_X=0;
 iCurrloc_Y=0;
 iFutloc_X=0;
 iFutloc_Y=0;
 iBoundary_X=0;
 iBoundary_Y=0;
 iMaxspeed=0;
 bMobility=false;
 iBeacontime=8;
 iLastbeacon=0;
 iNodeid=0;
 iPausetime=0;
 lOverhead =0;
 lMessage_succes =0;
 lMessage_drop =0;
 lMessage_sent =0;
 lTotaltrans =0;
 bCounted = false;
}

// Copy Constructor of the CNode class.

CNode::CNode(const CNode &x1)
{
        iPowerlevel = x1.iPowerlevel;
```

```
                iSecuritylevel= x1.iSecuritylevel;
                iSpeed = x1.iSpeed;
                iCurrloc_X = x1.iCurrloc_X;
                iCurrloc_Y = x1.iCurrloc_Y;
                iFutloc_X = x1.iFutloc_X;
                iFutloc_Y = x1.iFutloc_Y;
                iBoundary_X = x1.iBoundary_X;
                iBoundary_Y = x1.iBoundary_Y;
                iMaxspeed = x1.iMaxspeed;
                bMobility = x1.bMobility;
                iBeacontime = x1.iBeacontime;
                iLastbeacon = x1.iLastbeacon;
                iNodeid = x1.iNodeid;

                iPausetime = x1.iPausetime;
                lMessage_succes = x1.lMessage_succes;
                lMessage_drop = x1.lMessage_drop;
                lOverhead = x1.lOverhead;
                lMessage_sent = x1.lMessage_sent;
                lTotaltrans = x1.lTotaltrans;
                bCounted = x1.bCounted;
}

// The Equal to operator for the CNode class.

CNode& CNode::operator =(CNode x1)
{
   iPowerlevel = x1.iPowerlevel;
                iSecuritylevel = x1.iSecuritylevel;
                iSpeed = x1.iSpeed;
                iCurrloc_X = x1.iCurrloc_X;
                iCurrloc_Y = x1.iCurrloc_Y;
                iFutloc_X = x1.iFutloc_X;
                iFutloc_Y = x1.iFutloc_Y;
                iBoundary_X = x1.iBoundary_X;
                iBoundary_Y = x1.iBoundary_Y;
                iMaxspeed = x1.iMaxspeed;
                bMobility = x1.bMobility;
                iBeacontime = x1.iBeacontime;
                iLastbeacon = x1.iLastbeacon;
                iNodeid = x1.iNodeid;
                iPausetime = x1.iPausetime;
                lMessage_succes = x1.lMessage_succes;
                lMessage_drop = x1.lMessage_drop;
                lOverhead = x1.lOverhead;
                lMessage_sent = x1.lMessage_sent;
                lTotaltrans = x1.lTotaltrans;
                bCounted = x1.bCounted;
                return(*this);
}

//The function decides if a node should move to a new location.
//if it decides to move then it also decides the future location.

void CNode::Set_future(void)
{
 int iMobilityflag;
 iMobilityflag = rand()%3;
 if(iMobilityflag ==0)
 {
                if(iPausetime <= 0)
                {
                               bMobility = true; //Move to new location.
                               iSpeed = iMaxspeed; //Set speed to max speed.
                               iFutloc_X = rand()%iBoundary_X; //Set future locations.
                               iFutloc_Y = rand()%iBoundary_Y;
                }
                else
                {
                                iPausetime--; //Node waiting period.
                }
 }
```

```
 else
 {
              bMobility = false; //node stationary.
 }
}

// CNode class parameteric Constructor, used to intialize CNode
// class objects.

CNode::CNode(int Cx, int Cy, int Nid, int speed, int Bod_x, int Bod_y, long Power, long Beacon)
{
 iPowerlevel = Power;
 iCurrloc_X = Cx;
 iCurrloc_Y = Cy;
 iNodeid = Nid;
 iMaxspeed = speed;
 iBoundary_X = Bod_x;
 iBoundary_Y = Bod_y;
 bMobility = false;
 iBeacontime = Beacon;
 iLastbeacon =0;
 iSpeed =0;
 iFutloc_X =0;
 iFutloc_Y=0;
 iPausetime=0;
 lMessage_drop=0;
 lOverhead=0;
 lMessage_succes=0;
 lMessage_sent =0;
 bCounted = false;
}

//Moves the current location of the node near to its
//future location based on its speed. Checks the distance
//between current location and future location. if it is
//reachable in that instance then current location is
//changed to future location, else based on the speed
//the current location values are updated to a new location.
//After reaching the destination the mobility flag is turned off.

void CNode::Move_node(void)
{
        if(bMobility == true)
        {
        double theta;
        //calculating the distance from current location to destination
   double dTotaldistance=sqrt(double((iFutloc_X-iCurrloc_X)*(iFutloc_X-iCurrloc_X) + (iFutloc_Y-iCurrloc_Y)*(iFutloc_Y-
iCurrloc_Y)));
   double dDistancecovered = iSpeed;

        if(dDistancecovered >= dTotaldistance)
        {
    //if the node is in reachable distance from destination.
                iCurrloc_X = iFutloc_X;
                iCurrloc_Y = iFutloc_Y;
                bMobility = false; //Turns mobility flag off.
                iPausetime = rand()%25 + 5; //sets the pause time
        }
        else
        {
                if((iFutloc_X >=iCurrloc_X) && (iFutloc_Y >=iCurrloc_Y))
                {
                        //1st quadrant
                        if((iFutloc_X-iCurrloc_X)==0)
                                theta=3.142/2.0;
                        else
                                theta=atan(double((iFutloc_Y-iCurrloc_Y)/(iFutloc_X-iCurrloc_X)));
                        //Moves node to new location.
                        iCurrloc_X = iCurrloc_X + int(dDistancecovered * cos(theta));
                        iCurrloc_Y = iCurrloc_Y + int(dDistancecovered * sin(theta));
                }
                else if((iFutloc_X <= iCurrloc_X) && (iFutloc_Y >=iCurrloc_Y))
```

```
				{
					//2nd quadrant
					if((iFutloc_X-iCurrloc_X)==0)
						theta=3.142/2.0;
					else
						theta=atan(double((iFutloc_Y-iCurrloc_Y)/((-1)*(iFutloc_X-iCurrloc_X))));
					//Moves node to new location.
					iCurrloc_X = iCurrloc_X - int(dDistancecovered * cos(theta));
					iCurrloc_Y = iCurrloc_Y + int(dDistancecovered * sin(theta));
	}
				else if((iFutloc_X <= iCurrloc_X) && (iFutloc_Y <= iCurrloc_Y))
				{
					//3rd quadrant
					if((iFutloc_X-iCurrloc_X)==0)
						theta=3.142/2.0;
					else
						theta=atan(double(((-1)*(iFutloc_Y-iCurrloc_Y))/((-1)*(iFutloc_X-iCurrloc_X))));
					//Moves node to new location.
					iCurrloc_X=iCurrloc_X - int(dDistancecovered * cos(theta));
					iCurrloc_Y=iCurrloc_Y - int(dDistancecovered * sin(theta));
	}
				else if((iFutloc_X >= iCurrloc_X) && (iFutloc_Y <=iCurrloc_Y))
				{
					//4th quadrant
					if((iFutloc_X-iCurrloc_X)==0)
						theta=3.142/2.0;
					else
						 theta=atan(double(((-1)*(iFutloc_Y-iCurrloc_Y))/(iFutloc_X-iCurrloc_X)));
					//Moves node to new location.
					iCurrloc_X=iCurrloc_X + int(dDistancecovered * cos(theta));
					iCurrloc_Y=iCurrloc_Y - int(dDistancecovered * sin(theta));

	}

		}
		}
}

// Determines if a beacon needs to be sent. Returns true if yes.

bool CNode::Send_beacon(long iTime)
{
		if(bMobility==true)
		{
				//if the node is moving.
				if((iTime-iLastbeacon)>(iBeacontime-5))
				{
					iLastbeacon= iTime;
					iPowerlevel = iPowerlevel-5;
					return(true); //send beacon
				}
				else
				{
					return(false); //beacon not necessary.
				}
		}
		else
		{
				//if the node is stationary.
				if(((iTime-iLastbeacon)>iBeacontime)||(iTime<=5))
				{
					//Send beacon for the first 5 clock units, for the
					//network to be stable.
					iLastbeacon=iTime;
					iPowerlevel = iPowerlevel-5;
					return(true);
				}
				else
				{
					return(false);
				}
		}
```

```
}

/*
            File Name: AodvNode.h
            This file has the declaration of Ad hoc On-Demand Distance
            Vector(AODV) node. The class CAodvNode is used for this. The
            file also contains many other structures defined that are
            useful for the AODV node.
*/
# pragma once
# include "Node.h"

/*
    Class Structure CReverseroute is used as a table for storing
            the route information towards the source node. Each node has
            this table in it. An entry in this table is added or updated
            when ever a RREQ packet is recevied by a node. The table is
            used to send RRER packets to the source node.
*/
class CReverseroute
{
public:
  int iSourceid; //ID of the source node that issued RREQ.
  int iSequencenumber; //Source node sequence number.
  int iNumberofhops;//The distance from the source.
  int iNeighborid;//Neighbor from which RREQ was forwarded.
  long iLifetimestamp;//Time stamp when RREQ was received.
  int iRreqid; //The RREQ ID.
  //Constructors for supporting STL list class.
  CReverseroute(void);
  CReverseroute(const CReverseroute &x1);
  CReverseroute & operator= (CReverseroute x1);
  void Setreverseroute(int iSource, int iSequence, int iHops, int iNeighbor,int iRreqid, long iTime);
};

/*
            Class Structure CForwardroute is used as a table for storing
            the route information towards the destination node. All the
            nodes have this table. An entry in this table is added or
            updated when a RREP packet is received. The table is used
            to forward packets to the destination node.
*/
class CForwardroute
{
public:
            int iDestinationid; //Destination node ID.
            int iDestinsequenceno; //Destination sequence number.
            int iNeighborid; //Next Neighbor node id.
            int iNumberofhops; //Number of hops to destination.
            long iLifetimestamp;//Time stamp of RREP received.
            //Constructors supporting STL list class.
   CForwardroute(void);
            CForwardroute(const CForwardroute &x1);
            CForwardroute& operator= (CForwardroute x1);
            void Setforwardroute(int iDestination, int iNeighbor,int iDestseqno, int iHops, long iTime);

};

/*
            Class structure CAodvroutingtable is used for storing the
            routing information. Each node has this table to support
            the different data transmissions it is involved in.
*/
class CAodvroutingtable
{
public:
            int iSourceid; //Source node ID.
            int iDestinationid; //Destination node ID.
            int iSourceneighid; //Source side neighbor ID.
            int iDestinneighid; //Destination side neigbhor ID.
            long iLifetimestamp; //Time stamp of last use.
            //Constructors for supporting the STL list class.
```

```
		CAodvroutingtable(void);
		CAodvroutingtable(const CAodvroutingtable &x1);
		CAodvroutingtable& operator= (CAodvroutingtable x1);
		void Setroutingtable(int iSouid, int iDestinid, int iSouneighid, int iDestinneighid, long iTime);

};

/*
		The CAodvmessage class has multi-purpose use in this simulation.
	It is used as a message data structure- a medium to send messages
		between the nodes in the network. Each node has a list to store
		all the messages it receives and later reads them. The structure
		is also used as return type variable to exchange parameter values
		between the functions.
*/
class CAodvmessage
{
public:
		int iMessagetype; //The Message Type-like Beacon etc.
		int iSourceid; //Source node ID.
		int iDestinationid; //Destination node ID.
		int iSourcesequenceno; //Source sequence number.
		int iDestinationsequenceno; //Destination Sequence number.
		int iMessageid; //The Message ID, like RREQ ID.
		int iForwardnodeid; //Node that forwarded the message.
		int iNodeforwardtoid;//Message is forwarded to this node.
		long iTimetolive; //Time stamp.
		int iHopcount; //Hop count based on the packet.
		//Constructors to support the STL list class.
   CAodvmessage(void);
		CAodvmessage(const CAodvmessage &x1);
		//Operator Overload
		CAodvmessage& operator= (CAodvmessage x1);
		//Functions to create different packets.
		void CreateRREQ(int iSourid, int iSourceseq, int iDestinid, int iDestsequence, int iBroadid, int iHopc, long iTtl);
   void CreateRREP(int iSourid, int iDestinid, int iDestsequence, int iHopc, long iTtl, int iNodeforwardto);
		void CreateRERR(int iNeighborid,int iDestinid, int iNodeforwardto);
		void Setdatamessage(int iSourid, int iDestinid, int iMessid, int iForwardid, int iNodeforwardid);

};

/*
   The CAdajcency structure is used for maintaining the node neighbor
		information for each node. When a node receives a beacon from
		another node then an entry is added or updated in the adjacency
		table. Regular updates are made to have current neighbor information.
*/
class CAdjacency
{
public:
		int iNeighborid; //Neighbor Node ID.
		int iNeighsequence; //Neighbor sequence number.
		long iLifetime; //Time stamp of last beacon received.
		CAdjacency(void);
		CAdjacency(const CAdjacency &x1);
		CAdjacency & operator = (CAdjacency x1);
		void Setadjacency(int iNeighid, int iNeighseq, long iTime);
};

/*
		The class CAodvNode represents the AODV node in the network.
		It inherits the basic CNode class and has some additional data
		types and functions to support the AODV protocol features.
*/
class CAodvNode : public CNode
{
public:
		int iSequenceno; //Node Sequence number.
		int iCurrentrreqid; //Current RREQ ID to be assigned.
		int iNumberofpackets; //Number of packets to be transmitted.
		int iCurrentpacket; //Packets already tranmitted.
		int iCurrentdestin;//Current destination node ID.
```

```
		int iCurdesthopcount; //Number of hops to destination.
		bool bTransmitdata; //Whether transmitting data.
		bool bRouteest; //Whether route was established.
		bool bReqroute; //if route was requested.
		int iReqcount; //Number of times route was requested.
		long iInitialtime;//Time when route was requested.
	list <CReverseroute> listReverseroute; //Reserve routing table
		list <CForwardroute> listForwardroute; //Forward routing table
	list <CAodvroutingtable> listRoutingtable; //Routing information table.
		list <CAodvmessage> listMessage; //Message list.
		list <CAdjacency> listAdjacency; //Node adjacency list.
		vector <int> listGlobal; //Vector to support broadcasting.
		//Functions implemented in AodvNode.cpp file.
		void Add_adjacency(int iNeighid, int iNeighseq, long iTime);
	void Update_adjacency(long iTime);
		void Add_Reverseroute(CReverseroute Rroute);
		void Update_Reverseroute(long iTime);
		bool Add_Forwardroute(CForwardroute Froute);
		void Update_Forwardroute(long iTime);
		void Add_Routingtable(CAodvroutingtable Routingtable);
		void Update_Routingtable(long iTime);
		int Delete_Forwardroute(int iNeighborid);
		CAodvmessage Delete_Routingtable(int iNeighborid, int iDestid);
		bool Determine_Send_Messsage(long iTime, int iMaxnodes);
		CAodvmessage Send_Message(long iTime);
		CAodvmessage Read_Message(long iTime);
		CAodvmessage Verify_Rerr(long iTime);
		bool Check_Reverseroute(int iSourid, int iRid);
	int Neighbor_Reverseroute(int iSourid);
		int Delete_Forwardrouting(int iNeighid, int iDestid);
		void Updatetables(CAodvmessage Tempmessage, long iTime);
		void Updatealltables(long iTime);
		CAodvNode(void);
		CAodvNode(const CAodvNode &x1);
		CAodvNode & operator =(CAodvNode x1);
		void Forwardrreq(CAodvmessage Message);
};

/*
		File Name: AodvNode.cpp
		This file contains the implementation of functions present
		in the AodvNode.h header file. It has functions that are
		members of classes given in the AodvNode.h header file.
*/

# include "AodvNode.h"

// Default constructor for the CReverseroute class.

CReverseroute::CReverseroute(void)
{
		iSourceid =0;
		iSequencenumber =0;
		iNumberofhops =0;
		iNeighborid =0;
		iRreqid=0;
		iLifetimestamp =0;
}

// Copy constructor for the CReverseroute class.

CReverseroute::CReverseroute(const CReverseroute &x1)
{
		iSourceid = x1.iSourceid;
		iSequencenumber = x1.iSequencenumber;
		iNumberofhops = x1.iNumberofhops;
		iNeighborid = x1.iNeighborid;
		iRreqid = x1.iRreqid;
		iLifetimestamp = x1.iLifetimestamp;
}
// = operator Overloading.
```

```
CReverseroute & CReverseroute::operator =(CReverseroute x1)
{
		iSourceid = x1.iSourceid;
		iSequencenumber = x1.iSequencenumber;
		iNumberofhops = x1.iNumberofhops;
		iNeighborid = x1.iNeighborid;
		iRreqid = x1.iRreqid;
		iLifetimestamp = x1.iLifetimestamp;
		return(*this);
}
// Initates the CReverseroute object.

void CReverseroute::Setreverseroute(int iSource, int iSequence, int iHops, int iNeighbor,int iRid, long iTime)
{
		iSourceid = iSource;
		iSequencenumber = iSequence;
		iNumberofhops = iHops;
		iNeighborid = iNeighbor;
		iRreqid =iRid;
		iLifetimestamp = iTime;
}

// Default constructor for CForwardroute.

CForwardroute::CForwardroute(void)
{
		iDestinationid =0;
		iDestinsequenceno=0;
		iNeighborid =0;
		iNumberofhops =0;
		iLifetimestamp =0;
}

// Copy Constructor for CForwardroute.

CForwardroute::CForwardroute(const CForwardroute &x1)
{
		iDestinationid = x1.iDestinationid;
		iDestinsequenceno = x1.iDestinsequenceno;
		iNeighborid = x1.iNeighborid;
		iNumberofhops = x1.iNumberofhops;
		iLifetimestamp = x1.iLifetimestamp;
}

// = Operator Overloading.

CForwardroute& CForwardroute::operator=(CForwardroute x1)
{
		iDestinationid = x1.iDestinationid;
		iDestinsequenceno = x1.iDestinsequenceno;
		iNeighborid = x1.iNeighborid;
		iNumberofhops = x1.iNumberofhops;
		iLifetimestamp = x1.iLifetimestamp;
		return(*this);
}

// Initiates the CForwardroute Object with given values.

void CForwardroute::Setforwardroute(int iDestination, int iNeighbor,int iDestseqno, int iHops, long iTime)
{
		iDestinationid = iDestination;
		iNeighborid = iNeighbor;
		iDestinsequenceno = iDestseqno;
		iNumberofhops = iHops;
		iLifetimestamp = iTime;
}

// Default Constructor for CAodvroutingable class.

CAodvroutingtable::CAodvroutingtable(void)
{
		iSourceid =0;
```

```
        iDestinationid =0;
        iSourceneighid =0;
        iDestinneighid =0;
        iLifetimestamp =0;
}

// Copy Constructor for CAodvroutingtable class.

CAodvroutingtable::CAodvroutingtable(const CAodvroutingtable &x1)
{
        iSourceid = x1.iSourceid;
        iDestinationid = x1.iDestinationid;
        iSourceneighid = x1.iSourceneighid;
        iDestinneighid = x1.iDestinneighid;
        iLifetimestamp = x1.iLifetimestamp;
}

// = Operator Overloading.

CAodvroutingtable & CAodvroutingtable::operator =(CAodvroutingtable x1)
{
        iSourceid = x1.iSourceid;
        iDestinationid = x1.iDestinationid;
        iSourceneighid = x1.iSourceneighid;
        iDestinneighid = x1.iDestinneighid;
        iLifetimestamp = x1.iLifetimestamp;
        return(*this);
}

// Initiates the CAodvroutingtable class object to given values.

void CAodvroutingtable::Setroutingtable(int iSouid, int iDestindid, int iSouneighid, int iDestinneighid, long iTime)
{
  iSourceid = iSouid;
  iDestinationid = iDestindid;
  iSourceneighid = iSouneighid;
  iDestinneighid = iDestinneighid;
  iLifetimestamp = iTime;
}

// Default constructor for CAodvmessage class.

CAodvmessage::CAodvmessage(void)
{
        iMessagetype =0;
        iSourceid =0;
        iDestinationid =0;
        iSourcesequenceno =0;
        iDestinationsequenceno =0;
        iMessageid =0;
        iForwardnodeid =0;
        iNodeforwardtoid =0;
        iTimetolive =0;
        iHopcount =0;
}

// Copy Constructor for CAodvmessage class.

CAodvmessage::CAodvmessage(const CAodvmessage &x1)
{
        iMessagetype = x1.iMessagetype;
        iSourceid = x1.iSourceid;
        iDestinationid = x1.iDestinationid;
        iSourcesequenceno = x1.iSourcesequenceno;
        iDestinationsequenceno = x1.iDestinationsequenceno;
        iMessageid = x1.iMessageid;
        iForwardnodeid = x1.iForwardnodeid;
        iNodeforwardtoid = x1.iNodeforwardtoid;
        iTimetolive = x1.iTimetolive;
        iHopcount = x1.iHopcount;
}
```

```
// = Operator overloading.

CAodvmessage& CAodvmessage::operator= (CAodvmessage x1)
{
        iMessagetype = x1.iMessagetype;
        iSourceid = x1.iSourceid;
        iDestinationid = x1.iDestinationid;
        iSourcesequenceno = x1.iSourcesequenceno;
        iDestinationsequenceno = x1.iDestinationsequenceno;
        iMessageid = x1.iMessageid;
        iForwardnodeid = x1.iForwardnodeid;
        iNodeforwardtoid = x1.iNodeforwardtoid;
        iTimetolive = x1.iTimetolive;
        iHopcount = x1.iHopcount;
        return(*this);
}

// Function creates a Route request packet by initiating the CAodvmessage
// class object to the given values.

void CAodvmessage::CreateRREQ(int iSourid, int iSourceseq, int iDestinid, int iDestsequence, int iBroadid, int iHopc, long iTtl)
{
        iMessagetype = 1; //Route request messages are type 1 messages.
        iSourceid = iSourid;
        iSourcesequenceno = iSourceseq;
        iDestinationid = iDestinid;
        iDestinationsequenceno = iDestsequence;
        iMessageid = iBroadid;
        iHopcount = iHopc;
        iTimetolive = iTtl;

}

// Function creates a Route reply packet by initiating the CAodvmessage
// class object to the given values.

void CAodvmessage::CreateRREP(int iSourid, int iDestinid, int iDestsequence, int iHopc, long iTtl, int iNodeforwardto)
{
        iMessagetype = 2; //Route Reply is given message type 2.
        iSourceid = iSourid;
        iDestinationid = iDestinid;
        iDestinationsequenceno = iDestsequence;
        iHopcount = iHopc;
        iTimetolive = iTtl;
        iNodeforwardtoid = iNodeforwardto;
}
//Function creates a Route Error packet by initiating the CAodvmessage
//class object to the given values.
void CAodvmessage::CreateRERR(int iNeighborid, int iDestinid, int iNodeforwardto)
{
        iMessagetype =3;
        iSourceid = iNeighborid; //Temp storage of loss neighbor node id;
        iDestinationid = iDestinid;
        iNodeforwardtoid = iNodeforwardto;

}
//Function creates a data message packet by initiating the CAodvmessage
//class object to the given values.
void CAodvmessage::Setdatamessage(int iSourid, int iDestinid, int iMessid, int iForwardid, int iNodeforwardid)
{
        iMessagetype = 4; //Send Data to another node.
        iSourceid = iSourid;
        iDestinationid = iDestinid;
        iMessageid = iMessid;
        iForwardnodeid = iForwardid;
        iNodeforwardtoid = iNodeforwardid;
}

//Default constructor for CAdjacency class.
CAdjacency::CAdjacency(void)
{
        iNeighborid =0;
```

```
        iNeighsequence =0;
        iLifetime =0;
}
//Copy constructor for CAdjacency class.
CAdjacency::CAdjacency(const CAdjacency &x1)
{
        iNeighborid = x1.iNeighborid;
        iNeighsequence = x1.iNeighsequence;
        iLifetime = x1.iLifetime;
}
//= Operator Overloading.
CAdjacency & CAdjacency::operator=(CAdjacency x1)
{
        iNeighborid = x1.iNeighborid;
        iNeighsequence = x1.iNeighsequence;
        iLifetime = x1.iLifetime;
        return(*this);
}
//Function initiates the CAdjacency class object to given values.
void CAdjacency::Setadjacency(int iNeighid, int iNeighseq, long iTime)
{
        iNeighborid = iNeighid;
        iNeighsequence = iNeighseq;
        iLifetime = iTime;
}
/*
        The function creates a CAdjacency object and adds it to the
        Adjacency list. This function is called when a beacon is received
        by the node. if the neighbor information is already present in the
        list then it is updated else a new entry is added.
*/
void CAodvNode::Add_adjacency(int iNeighid, int iNeighseq, long iTime)
{
        bool bFlag;
        bFlag = true;
        list <CAdjacency>::iterator itPointer;
        //Removing the previous instance of the neighbor information.
        for(itPointer = listAdjacency.begin(); itPointer != listAdjacency.end(); itPointer++)
        {
                CAdjacency Curr_node = *itPointer;
                if(Curr_node.iNeighborid == iNeighid)
                {
                        listAdjacency.erase(itPointer);
                        bFlag = false;
                        break;
                }
        }
  //Add the current instance of the neighbor information.
        CAdjacency Curr_node;
        Curr_node.Setadjacency(iNeighid, iNeighseq, iTime);
        listAdjacency.push_back(Curr_node);
        if(bFlag ==true)
        {
                //Change sequence number as adjacency structure is changed.
                iSequenceno++;
        }
}
/*
        The function is used for storing current information in the
        adjacency list. Outdated information is regularly removed from
        the list. if an entry in the adjacency is timed out then it is
        removed. Different timeout values are used depending on nodes
        mobility.
*/
void CAodvNode::Update_adjacency(long iTime)
{
        long iTemptime;
        bool bFlag;
        bFlag = false;
        if(bMobility == true)
        {
                //if the node is in motion.
```

```
			iTemptime = 10;
		}
		else
		{
			//if the node is stationary.
			iTemptime = 15;
		}
		list <CAdjacency>::iterator itPointer;
		list <CAdjacency> listTemp;
		//Transfer the objects into a temprorary list.
		while(!listAdjacency.empty())
		{
			itPointer = listAdjacency.begin();
			CAdjacency Curr_node = *itPointer;
			if((iTime - Curr_node.iLifetime)<= iTemptime)
			{
				//Valid time stamp entry.
				listTemp.push_back(Curr_node);
			}
			else
			{
				//Entry is timed out and is removed from list.
				bFlag = true;
			}
			listAdjacency.pop_front();
		}
		//objects in temp list are moved to adjacency list.
		while(!listTemp.empty())
		{
			itPointer = listTemp.begin();
			CAdjacency Curr_node = *itPointer;
			listAdjacency.push_back(Curr_node);
			listTemp.pop_front();

		}
		if(bFlag == true)
		{
			//Detected change in environment so sequence number
			// is incremented.
			iSequenceno++;
		}
}
/*
		The function is used to add a CReverseroute object into
		the Reverseroute list. Generally called when a RREQ is
		received by the node.
*/
void CAodvNode::Add_Reverseroute(CReverseroute Rroute)
{
		bool bFlag;
		bFlag= false;
		list <CReverseroute>::iterator itPointer;
		//Removing any previous information.
		for(itPointer = listReverseroute.begin(); itPointer != listReverseroute.end(); itPointer++)
		{
			CReverseroute Curr_node= *itPointer;
			if(Curr_node.iSourceid == Rroute.iSourceid)
			{
				if(Curr_node.iNumberofhops >= Rroute.iNumberofhops)
				{
					//New route is the shortest one.
					listReverseroute.erase(itPointer);
					break;
				}
				else
				{
					//Old route was the shortest one.
					bFlag = true;
					break;
				}
			}
		}
```

```
		if(bFlag == false)
		{
			//Add a new entry into the list.
			listReverseroute.push_back(Rroute);
		}
}
/*
		The functions maintians only the current information in
		the reverseroute list. It removes all entries that are
		timed out.
*/
void CAodvNode::Update_Reverseroute(long iTime)
{
		list <CReverseroute>::iterator itPointer;
		list <CReverseroute> listTemp;
		//Move entries into temp list.
		while(!listReverseroute.empty())
		{
			itPointer = listReverseroute.begin();
			CReverseroute Curr_node = *itPointer;
			if((iTime-Curr_node.iLifetimestamp)<= 5)
			{
				//if entries are not timed out.
				listTemp.push_back(Curr_node);
			}
			listReverseroute.pop_front();
		}
		//Move from temp list to reverseroute list.
		while(!listTemp.empty())
		{
			itPointer = listTemp.begin();
			CReverseroute Curr_node = *itPointer;
			listReverseroute.push_back(Curr_node);
			listTemp.pop_front();
		}

}
/*
		The function is used to add a CForwardroute object into
		the Forwardroute list. Generally called when a RREP is
		received by the node.
*/
bool CAodvNode::Add_Forwardroute(CForwardroute Froute)
{
		bool bFlag;
		bFlag = false;
		list <CForwardroute>::iterator itPointer;
		//Check is the entry is already present.
		for(itPointer= listForwardroute.begin(); itPointer != listForwardroute.end(); itPointer++)
		{
			CForwardroute Curr_node = *itPointer;
			//Match with destination.
			if((Curr_node.iDestinationid == Froute.iDestinationid)&&(Curr_node.iDestinsequenceno <=
Froute.iDestinsequenceno))
			{
				if(Curr_node.iNumberofhops >= Froute.iNumberofhops)
				{
					//When the new route has fewer number of hops.
					listForwardroute.erase(itPointer);
					break;
				}
				else
				{
					//The previous route is used.
					bFlag=true;
					break;
				}
			}

		}
		if(bFlag == false)
		{
```

```
                        //when the entry is added.
                        listForwardroute.push_back(Froute);
                        return(true);
            }
            else
            {
                        //When the entry is not added.
                        return(false);
            }
}
/*
            The functions maintians only the current information in
            the Forwardroute list. It removes all entries that are
            timed out.
*/
void CAodvNode::Update_Forwardroute(long iTime)
{
            list <CForwardroute>::iterator itPointer;
            list <CForwardroute> listTemp;
            //Move objects to temp list.
            while(!listForwardroute.empty())
            {
             itPointer = listForwardroute.begin();
             CForwardroute Curr_node = *itPointer;
             if((iTime-Curr_node.iLifetimestamp) <= 5)
             {
                         //Valid entries- not timed out.
                         listTemp.push_back(Curr_node);
             }
             listForwardroute.pop_front();
            }
            //Move from temp list to forwardroute list.
            while(!listTemp.empty())
            {
                        itPointer = listTemp.begin();
                        CForwardroute Curr_node = *itPointer;
                        listForwardroute.push_back(Curr_node);
                        listTemp.pop_front();
            }

}
/*
 The function adds a CAodvroutingtable object to the Routingtable list
 if the entity is already present in the table then its values are
 updated.
*/
void CAodvNode::Add_Routingtable(CAodvroutingtable Routingtable)
{
            list <CAodvroutingtable>::iterator itPointer;
            //Check if object is already present.
            for(itPointer = listRoutingtable.begin(); itPointer != listRoutingtable.end(); itPointer++)
            {
                        CAodvroutingtable Curr_node = *itPointer;
                        if((Curr_node.iDestinationid== Routingtable.iDestinationid)&&(Curr_node.iSourceid== Routingtable.iSourceid))
                        {
                                    //Delete the already present object.
                                    listRoutingtable.erase(itPointer);
                                    break;
                        }
            }
            //Add the object to the list.
            listRoutingtable.push_back(Routingtable);
}
/*
            Update the routing table similar to the other tabels.
*/
void CAodvNode::Update_Routingtable(long iTime)
{
            list <CAodvroutingtable>::iterator itPointer;
            list <CAodvroutingtable> listTemp;
            //Move to temp list.
            while(!listRoutingtable.empty())
```

```
		{
			itPointer= listRoutingtable.begin();
			CAodvroutingtable Curr_node = *itPointer;
			if((iTime-Curr_node.iLifetimestamp)<= 5)
			{
				// Non timed out entry is mvoe to temp list.
				listTemp.push_back(Curr_node);
			}
			listRoutingtable.pop_front();
		}
		//Move from temp list to Routingtable list.
		while(!listTemp.empty())
		{
			itPointer = listTemp.begin();
			CAodvroutingtable Curr_node = *itPointer;
   listRoutingtable.push_back(Curr_node);
			listTemp.pop_front();
		}
}

/*
		The function Determine_Send_Message decides if a node
		wants to transmit data to another node. it has the current
		clock value and maximum number of nodes present as inputs.
		if a node randomly decides to transmit data then the
		function returns true else false. The function also decides
		the destination node and the number of packets to be
		transmitted.
*/
bool CAodvNode::Determine_Send_Messsage(long iTime, int iMaxnodes)
{
		if(bTransmitdata == false)
		{
			//Currently the node is not transmitting.
		int iProb, iTempnode, iMul;
		iProb = rand()%10;
   if(iProb < 8)
		{
			//When no data is to be sent
			return(false);
		}
		else
		{
			 //When data is to be sent.
     iTempnode = rand()%iMaxnodes; //Determine the destination id.
		 //Check that the node id does not go out of bound.
		 if(iTempnode == iNodeid)
		 {
			 if(iTempnode < (iMaxnodes-2))
			 {
				 iTempnode++;
			 }
			 else
			 {
				 if(iTempnode==0)
				 {
					 iTempnode++;
				 }
				 else
				 {
					 iTempnode--;
				 }
			 }
		 }
		 iCurrentdestin = iTempnode;
		 //Determine the number of packets to be transmitted.
		 iMul = rand()%10;
		 iMul++;
		 iNumberofpackets = 100* iMul;
		 //Reset the variables.
		 bRouteest = false;
		 bReqroute=false;
```

```
                bTransmitdata = true;
                iCurrentpacket =0;
                lTotaltrans++;
                return(true);

            }
            }
            return(false);
}
/*
            The function Send_Message decides the packet to be sent. it has
            the current simulator clock value as the input. It returns a
            CAodvmessage that is read by network layer functions. The function
            decides whether to broadcast a RREQ packet. Once the route is
            established it transmits the packets one after the other. The
            function send a maximum of 3 RREQ packets for each transmission.
            if all three RREQ packets are timed out then transmission is
            aborted.
*/
CAodvmessage CAodvNode::Send_Message(long iTime)
{
            CAodvmessage Returnmessage;
            if(bTransmitdata == true)
            {
                        //When the node has data to transmit.
            Returnmessage.iDestinationid = iCurrentdestin;
            Returnmessage.iSourceid = iNodeid;
            Returnmessage.iSourcesequenceno = iSequenceno;
            Returnmessage.iTimetolive = iTime;
            if(bRouteest==false)
            {
                        //When the destination should be located.
                        if(bReqroute==false)
                        {
                                    //When the route was not previously requested.
                                    list <CAdjacency>::iterator itAdjpointer;
                                    Returnmessage.iMessageid =iCurrentpacket;
                                    bReqroute = true;
                                    for(itAdjpointer = listAdjacency.begin(); itAdjpointer != listAdjacency.end(); itAdjpointer++)
                                    {
                                                CAdjacency Tempadj = *itAdjpointer;
                                                if(Tempadj.iNeighborid == iCurrentdestin)
                                                {
                                                            //When destination is a neighbor node.
                                                            Returnmessage.iDestinationsequenceno = Tempadj.iNeighsequence;
                                                            iCurdesthopcount=0;
                                                            Returnmessage.iHopcount =0;
                                                            bRouteest = true;
                                                            CForwardroute Tempforwardroute;
                                                            Tempforwardroute.iDestinationid = iCurrentdestin;
                                                            Tempforwardroute.iDestinsequenceno = Tempadj.iNeighsequence;
                                                            Tempforwardroute.iLifetimestamp = iTime;
                                                            Tempforwardroute.iNeighborid = iCurrentdestin;
                                                            Tempforwardroute.iNumberofhops =1;
                                                            Add_Forwardroute(Tempforwardroute);

            Returnmessage.Setdatamessage(iNodeid,iCurrentdestin,iCurrentpacket,iNodeid,iCurrentdestin);
                                                            iPowerlevel = iPowerlevel -250; //Reduce power level
                                                            //Start Data transmission.
                                                            return(Returnmessage);
                                                }
                                    }
                                    list <CForwardroute>::iterator itForpointer;
                                    for(itForpointer = listForwardroute.begin(); itForpointer != listForwardroute.end(); itForpointer++)
                                    {
                                                CForwardroute Tempfor = *itForpointer;
                                                if(Tempfor.iDestinationid == iCurrentdestin)
                                                {
                                                             //When destination is present in forward routes.
                                                            Returnmessage.iHopcount = Tempfor.iNumberofhops;
                                                            iCurdesthopcount = Tempfor.iNumberofhops;
                                                            bRouteest = true;
```

```
Returnmessage.Setdatamessage(iNodeid,iCurrentdestin,iCurrentpacket,iNodeid,Tempfor.iNeighborid);
                                                        iPowerlevel = iPowerlevel -250;
                                                        lMessage_sent++;
                                                        //As route already established so start transmission.
                                                        return(Returnmessage);
                                                }
                                        }

                                        Returnmessage.iMessagetype = 1; //Requesting a route
                                        iInitialtime= iTime;
                                        iReqcount =0;
                                        Returnmessage.CreateRREQ(iNodeid,iSequenceno,iCurrentdestin,0,iCurrentrreqid,0,iTime);
                                        Returnmessage.iForwardnodeid = iNodeid;
                                        Returnmessage.iNodeforwardtoid = -1; // Initiated from here.
                                        iPowerlevel = iPowerlevel-75;
                                        lOverhead++;
                                        //Route is not present sending a RREQ packet.
return(Returnmessage);
                                }
                                else
                                {
if((iTime - iInitialtime)< 10)
                                 {
                                        //Waiting for the route to be established.
                                        Returnmessage.iMessagetype =0; //Do nothing.
                                        return(Returnmessage);
                                 }
                                 else
                                 {
                                        if(iReqcount <3)
                                        {
                                                //if the route is requested less than 3 times.
                                                        Returnmessage.iMessagetype =1; //Requesting a route  again.

Returnmessage.CreateRREQ(iNodeid,iSequenceno,iCurrentdestin,0,iCurrentrreqid,0,iTime);
                                                        Returnmessage.iForwardnodeid = iNodeid;
                                                        Returnmessage.iNodeforwardtoid = -1; //Initiated from here.
                                                        iPowerlevel = iPowerlevel -75;
                                                        lOverhead++;
                                                        iCurrentrreqid++;
                                                        iReqcount++;
                                                        iInitialtime = iTime;
                                                        return(Returnmessage);
                                        }
                                        else
                                        {
                                                //Route requested more than 3 times.
                                                        bTransmitdata = false; //Cancel the data transmission.
                                                        bRouteest = false;
                                                        bReqroute = false;
                                                        Returnmessage.iMessagetype =0; //Do nothing
                                                        return(Returnmessage);
                                        }
                                 }
                                }
        }
        else
        {
                //The routes have been established.
if(iCurrentpacket < iNumberofpackets)
         {
                  //Packets are present to transmit.
                iCurrentpacket++;
                Returnmessage.iMessagetype = 4; //send data packet.
                Returnmessage.iMessageid = iCurrentpacket;
                Returnmessage.iHopcount = iCurdesthopcount;
                Returnmessage.iForwardnodeid = iNodeid;
                list <CForwardroute>::iterator itForpointer;
                //Obtain the forward route information.
                        for(itForpointer = listForwardroute.begin(); itForpointer != listForwardroute.end(); itForpointer++)
                        {
```

```
                                        CForwardroute Tempfor = *itForpointer;
                                        if(Tempfor.iDestinationid == iCurrentdestin)
                                        {
                                                Returnmessage.Setdatamessage(iNodeid,
iCurrentdestin,iCurrentpacket,iNodeid,Tempfor.iNeighborid);
                                                Updatetables(Returnmessage, iTime);
                                                iPowerlevel = iPowerlevel-250;
                                                lMessage_sent++;
                                                return(Returnmessage);
                                        }
                                }
                        //Did'nt find the forward route.
                        bTransmitdata =false;
                        bRouteest = false;
                        bReqroute =false;
                        Returnmessage.iMessagetype=0; //Do nothing.
    return(Returnmessage);
                }
                else
                {
                        //Completed data transmission.
                        bTransmitdata =false;
                        bRouteest = false;
                        bReqroute =false;
                        Returnmessage.iMessagetype = 0; //Do nothing.
    return(Returnmessage);
                }

        }
        }
        Returnmessage.iMessagetype =0; //Do nothing.
        return(Returnmessage);
}
/*
        The function to remove an entry from forward route. The
        neigbhor ID is used for identifying the object and the
        corresponding destination id is returned.
*/
int CAodvNode::Delete_Forwardroute(int iNeighborid)
{
        list <CForwardroute>::iterator itForpointer;
        //Identifying the Object.
        for(itForpointer = listForwardroute.begin(); itForpointer != listForwardroute.end(); itForpointer++)
        {
                CForwardroute Temproute = *itForpointer;
                if(Temproute.iNeighborid == iNeighborid)
                {
                        listForwardroute.erase(itForpointer);
                        return(Temproute.iDestinationid); //returns the destination id.

                }
        }
        return(-1); //if the object is not present.
}
/*
        The function removes an entry from Routing table. The neighbor ID
        and Destination ID are used to trace the instance. It returns the
        information about the source id, destination id, source side
        neighbor id of that instance.
*/
CAodvmessage CAodvNode::Delete_Routingtable(int iNeighborid, int iDestid)
{
        CAodvmessage Returnmessage;
        list <CAodvroutingtable>::iterator itRoutingtable;
        for(itRoutingtable = listRoutingtable.begin(); itRoutingtable != listRoutingtable.end(); itRoutingtable++)
        {
                CAodvroutingtable Temptable = *itRoutingtable;
                if((Temptable.iDestinneighid == iNeighborid)&&(Temptable.iDestinationid == iDestid))
                {
                        Returnmessage.iMessagetype = 1; //Sucessful location.
     Returnmessage.iSourceid = Temptable.iSourceid;
                        Returnmessage.iDestinationid = Temptable.iDestinationid;
```

```
					Returnmessage.iNodeforwardtoid = Temptable.iSourceneighid;
					Returnmessage.iHopcount = -1; //The following destination unreachable.
					listRoutingtable.erase(itRoutingtable);
					return(Returnmessage);
				}
		}
		Returnmessage.iMessagetype=0; //Unsucessful location.
		return(Returnmessage);

}
/*
		This function is used to identify any loss in links at
		that node. When it detects the loss of a current route
		then it issues a RERR message to the source node.
*/
CAodvmessage CAodvNode::Verify_Rerr(long iTime)
{
		list <CForwardroute>::iterator itForpointer;
		CAodvmessage Returnmessage, Tempmessage;
		bool bFlag;
		bFlag = false;
		list <CAdjacency>::iterator itAdjpointer;
		//Check the forward route neighbors with the adjacency list.
		for(itForpointer = listForwardroute.begin(); itForpointer != listForwardroute.end(); itForpointer++)
		{
				CForwardroute Tempforward = *itForpointer;
				for(itAdjpointer= listAdjacency.begin(); itAdjpointer != listAdjacency.end(); itAdjpointer++)
				{
					CAdjacency Tempadj = *itAdjpointer;
					if(Tempforward.iNeighborid == Tempadj.iNeighborid)
					{
						bFlag = true;
						break;
					}

				}
				if(bFlag==false)
				{
					//The node was detected missing.
					int iTempdest;
					Returnmessage.iMessagetype =0; // Issue a Rerr.
					iTempdest= Delete_Forwardroute(Tempforward.iNeighborid);
					if(iTempdest !=-1)
					{
					Tempmessage = Delete_Routingtable(Tempforward.iNeighborid, iTempdest);
					if(Tempmessage.iSourceid == iNodeid)
					{
						bTransmitdata =false;
						bRouteest = false;
						bReqroute = false;
						Returnmessage.iMessagetype =1; //Do nothing.
						return(Returnmessage);
					}
	Returnmessage.iSourceid = Tempmessage.iSourceid;
					Returnmessage.iDestinationid = Tempmessage.iDestinationid;
					Returnmessage.iNodeforwardtoid = Tempmessage.iNodeforwardtoid;
					Returnmessage.iHopcount = Tempmessage.iHopcount;
					return(Returnmessage);
					}
				}
		}
		Returnmessage.iMessagetype =1; //Do nothing.
		return(Returnmessage);
}
/*
		The funciton checks wethere an entry in the Reverse route
		table is already present. The source node id and the RREQ
		ID are used to check. Returns true if present already.
*/
bool CAodvNode::Check_Reverseroute(int iSourid, int iRid)
{
		bool bFlag;
```

```
		list <CReverseroute>::iterator itReverse;
		bFlag=false;
		for(itReverse = listReverseroute.begin(); itReverse != listReverseroute.end(); itReverse++)
		{
			CReverseroute Tempreverse = *itReverse;
			if((Tempreverse.iSourceid == iSourid)&&(Tempreverse.iRreqid == iRid))
			{
				//The Instance already present in table.
				bFlag=true;
				break;
			}
		}
		return(bFlag);
}
/*
		The function it retruns the Neighborn node ID from which
		it received RREQ message of the given source node.
*/
int CAodvNode::Neighbor_Reverseroute(int iSourid)
{
		list <CReverseroute>::iterator Tempreverse;
		for(Tempreverse= listReverseroute.begin(); Tempreverse != listReverseroute.end(); Tempreverse++)
		{
			CReverseroute Reverseroute =*Tempreverse;
			if(Reverseroute.iSourceid == iSourid)
			{
				//Identified the entry.
				return(Reverseroute.iNeighborid);
			}
		}
		return(-1); //Entry missing corresponding to source.
}
/*
		The function is used to delete an entry from the
		forward routing table. The entry corresponding to
		the given Neighbor ID and Destination ID is deleted.
*/
int CAodvNode::Delete_Forwardrouting(int iNeighid, int iDestid)
{
		list <CForwardroute>::iterator itForward;
		//Identify and delete entry from table.
		for(itForward= listForwardroute.begin(); itForward != listForwardroute.end(); itForward++)
		{
			CForwardroute Tempforward =*itForward;
			if((Tempforward.iDestinationid == iDestid)&&(Tempforward.iNeighborid == iNeighid))
			{
				listForwardroute.erase(itForward);
				break;
			}
		}
		list <CAodvroutingtable>::iterator itRouting;
		//Getting the source side neighbor information from routing table.
		for(itRouting=listRoutingtable.begin(); itRouting!= listRoutingtable.end(); itRouting++)
		{
			CAodvroutingtable Temproute = *itRouting;
			if((Temproute.iDestinationid==iDestid)&&(Temproute.iDestinneighid == iNeighid))
			{
				listRoutingtable.erase(itRouting);
				return(Temproute.iSourceneighid);
			}
		}
		return(-1);
}

/*
		The function is used to update the time stamp an entry
		present in all the three tables. This function is called
		when a current route is used for data transmission.
*/
void CAodvNode::Updatetables(CAodvmessage Tempmessage, long iTime)
{
  //Initially updating the Reverse route.
```

```
		list <CReverseroute>::iterator itReverse;
		for(itReverse = listReverseroute.begin(); itReverse != listReverseroute.end(); itReverse++)
		{
			CReverseroute Temproute = *itReverse;
			if(Temproute.iSourceid == Tempmessage.iSourceid)
			{
				Temproute.iLifetimestamp = iTime;
				listReverseroute.erase(itReverse);
				listReverseroute.push_back(Temproute);
				break;
			}
		}
	//Updating the entry in Forward route.
		list<CForwardroute>::iterator itFor;
		for(itFor = listForwardroute.begin(); itFor != listForwardroute.end(); itFor++)
		{
			CForwardroute Tempfor = *itFor;
			if(Tempfor.iDestinationid == Tempmessage.iDestinationid)
			{
				Tempfor.iLifetimestamp = iTime;
				listForwardroute.erase(itFor);
				listForwardroute.push_back(Tempfor);
				break;
			}
		}
		//Updating the routing table entry.
		list <CAodvroutingtable>::iterator itRoute;
		for(itRoute= listRoutingtable.begin(); itRoute != listRoutingtable.end(); itRoute++)
		{
			CAodvroutingtable Temproute = *itRoute;
			if((Temproute.iDestinationid == Tempmessage.iDestinationid) && (Temproute.iSourceid ==
Tempmessage.iSourceid))
			{
				Temproute.iLifetimestamp = iTime;
				listRoutingtable.erase(itRoute);
				listRoutingtable.push_back(Temproute);
				break;
			}
		}
}
/*
		This function is one of the basic repsonsibilities of a node.
		Function is used to read the messages it has received. The messages
		are present in a queue and it reads them one at a time. After reading
		a message the node has to perform an action appropriately. The
		function is called until the list is empty or message time stamp is
		less than the current clock time.
*/
CAodvmessage CAodvNode::Read_Message(long iTime)
{
		CAodvmessage Returnmessage;
		bool bFlag;
		if(!listMessage.empty())
		{
			//The list is not empty.
			if(iPowerlevel <=0)
			{
				//if the node does not have enough power resources.
				listMessage.clear();
				Returnmessage.iMessagetype=6; //No need to read.
				return(Returnmessage);
			}
			CAodvmessage Topmessage = listMessage.front();
			if(Topmessage.iTimetolive < iTime)
			{
				//Message to be read during this cycle.
				listMessage.pop_front();
				if((iTime - Topmessage.iTimetolive)>10)
				{
					//Message time stamp expired.
					iPowerlevel = iPowerlevel - 5;
					Returnmessage.iMessagetype=5; //Ignore message.
```

```
				return(Returnmessage);
			}
			if(Topmessage.iNodeforwardtoid == iNodeid)
			{
				//Node is the intended receiver of message.
				if(Topmessage.iMessagetype ==0) //Received a beacon from a node.
				{
					Add_adjacency(Topmessage.iSourceid,Topmessage.iSourcesequenceno, iTime);
					iPowerlevel = iPowerlevel -5;
					Returnmessage.iMessagetype=5;
					//Beacon nothing to be done, ignore message.
					return(Returnmessage);
				}
				if(Topmessage.iMessagetype ==1)
				{
					//Received Route request packet.
					bool bRedundancy;
					iPowerlevel= iPowerlevel - 50;
					bRedundancy = Check_Reverseroute(Topmessage.iSourceid,
Topmessage.iMessageid);
					if(bRedundancy==true)
					{
						Returnmessage.iMessagetype=5;// Ignore RREQ Message.
						return(Returnmessage);
					}
					CReverseroute Tempreverse;
					Tempreverse.iLifetimestamp = iTime;
					Tempreverse.iNeighborid = Topmessage.iForwardnodeid;
					Tempreverse.iNumberofhops = Topmessage.iHopcount +1;
					Tempreverse.iSequencenumber = Topmessage.iSourcesequenceno;
					Tempreverse.iSourceid = Topmessage.iSourceid;
					Tempreverse.iRreqid = Topmessage.iMessageid;
					Add_Reverseroute(Tempreverse);
					//Incase the current node is the destination requested.
					if(Topmessage.iDestinationid == iNodeid)
					{
						CAodvroutingtable Temprouting;
						Temprouting.iDestinationid = Topmessage.iDestinationid;
						Temprouting.iDestinneighid = Topmessage.iDestinationid;
						Temprouting.iLifetimestamp = iTime;
						Temprouting.iSourceid = Topmessage.iSourceid;
						Temprouting.iSourceneighid = Topmessage.iForwardnodeid;
						Add_Routingtable(Temprouting);
						//Route reply packets.
						Returnmessage.CreateRREP(Topmessage.iSourceid,
Topmessage.iDestinationid, iSequenceno,0, iTime,Topmessage.iForwardnodeid);
						Returnmessage.iForwardnodeid = iNodeid;
						return(Returnmessage);
					}
					else
					{
						//Checking if the destination is present in the current routing table.
						list <CForwardroute>::iterator itForward;
						bFlag = false;
						CForwardroute Tempforroute;
						for(itForward = listForwardroute.begin(); itForward !=
listForwardroute.end(); itForward++)
						{
							Tempforroute = *itForward;
							if((Tempforroute.iDestinationid ==
Topmessage.iDestinationid)&&(Tempforroute.iDestinsequenceno <= Topmessage.iDestinationsequenceno))
							{
							 bFlag = true;
							 break;
							}
						}
						if(bFlag==true)
						{
							//the Route is available sending a Route reply.
							CAodvroutingtable Temptable;
							Temptable.Setroutingtable(Topmessage.iSourceid,
Topmessage.iDestinationid, Topmessage.iForwardnodeid, Tempforroute.iNeighborid, iTime);
```

```
									Add_Routingtable(Temptable);
									//Entry in routing table.
									//Creating a Route reply packet.
									Returnmessage.CreateRREP(Topmessage.iSourceid,
Topmessage.iDestinationid,Tempforroute.iDestinsequenceno,Tempforroute.iNumberofhops, iTime, Topmessage.iForwardnodeid);
									Returnmessage.iForwardnodeid = iNodeid;
									return(Returnmessage);
								}
								else
								{
									//The Route is not avialable in the routing table.
									iPowerlevel = iPowerlevel - 75;
									Returnmessage.CreateRREQ(Topmessage.iSourceid,
Topmessage.iSourcesequenceno, Topmessage.iDestinationid, Topmessage.iDestinationsequenceno,
Topmessage.iMessageid,(Topmessage.iHopcount+1),iTime);
									Returnmessage.iForwardnodeid = iNodeid;
									Returnmessage.iNodeforwardtoid =
Topmessage.iForwardnodeid; //not to be forwarded to this id.
									lOverhead++;
									return(Returnmessage);
								}
							}
						}
						if(Topmessage.iMessagetype==2)
						{
							//Received Route Reply message.
							CForwardroute Temproute;
							CAodvroutingtable Temptable;
							if(Topmessage.iSourceid == iNodeid)
							{
								//When the source node receives the Route reply message.
								bool bUpdatedflag;
								Temproute.Setforwardroute(Topmessage.iDestinationid,
Topmessage.iForwardnodeid, Topmessage.iDestinationsequenceno, (Topmessage.iHopcount+1), iTime);
								bUpdatedflag = Add_Forwardroute(Temproute);
								if(bUpdatedflag==true)
								{

	Temptable.Setroutingtable(iNodeid,iCurrentdestin,iNodeid,Topmessage.iForwardnodeid,iTime);
									Add_Routingtable(Temptable);
								}
								bRouteest=true;
								Returnmessage.iMessagetype= 5; //Do nothing.
								return(Returnmessage);
							}
							else
							{
								//When an intermediate node receives the RREP message.

	Temproute.Setforwardroute(Topmessage.iDestinationid,Topmessage.iForwardnodeid,
Topmessage.iDestinationsequenceno,(Topmessage.iHopcount+1), iTime);
								Add_Forwardroute(Temproute);
								int iTempneighborid;
								iTempneighborid = Neighbor_Reverseroute(Topmessage.iSourceid);
								if(iTempneighborid==-1)
								{
									Returnmessage.iMessagetype=5;
									//Do nothing, as entry expired.
									return(Returnmessage);
								}
								CAodvroutingtable Temproutingtable;

	Temproutingtable.Setroutingtable(Topmessage.iSourceid,Topmessage.iDestinationid,iTempneighborid,Topmessage.iForwardn
odeid,iTime);
								Add_Routingtable(Temproutingtable);
								Returnmessage.CreateRREP(Topmessage.iSourceid,
Topmessage.iDestinationid, Topmessage.iDestinationsequenceno,(Topmessage.iHopcount+1),iTime,iTempneighborid);
								Returnmessage.iTimetolive = Topmessage.iTimetolive;
								Returnmessage.iForwardnodeid = iNodeid;
								return(Returnmessage);
							}
						}
```

```
if(Topmessage.iMessagetype==3)
{
	//Received a RERR Message.
	if((Topmessage.iDestinationid  == iCurrentdestin)&&(bTransmitdata==true))//
if the source recevies RERR message.
	{
		bTransmitdata = false;
		bRouteest = false;
		bReqroute = false;
		iReqcount=0;
		Returnmessage.iMessagetype =5; //Do nothing.
		return(Returnmessage);
	}
	else
	{
		//if an intermediate node receives the RERR message.
		int iTempsourceneigh;
		iTempsourceneigh =
Delete_Forwardrouting(Topmessage.iSourceid,Topmessage.iDestinationid);
		if(iTempsourceneigh==-1)
		{
			Returnmessage.iMessagetype =5; //Do nothing.
			return(Returnmessage);
		}
		Returnmessage.CreateRERR(iNodeid,Topmessage.iDestinationid,
iTempsourceneigh);
		Returnmessage.iTimetolive = Topmessage.iTimetolive;
		Returnmessage.iForwardnodeid = iNodeid;
		return(Returnmessage);
	}
}
if(Topmessage.iMessagetype ==4)
{
	//Received a Data Packet/Message.
	iPowerlevel = iPowerlevel - 150;
	Updatetables(Topmessage,iTime);
	if(Topmessage.iDestinationid ==iNodeid)
	{
		//The packet receved destination.
		Returnmessage.iMessagetype = 5;
		lMessage_succes++;
		return(Returnmessage);
	}
	else
	{
		//Intermediate node has received the data packet.
		list <CForwardroute>::iterator itTempforward;
		bool bForwardflag;
		bForwardflag=false;
		CForwardroute Tempforroute;
		//Locating the route in the forward table.
		for(itTempforward= listForwardroute.begin(); itTempforward!=
listForwardroute.end(); itTempforward++)
		{
			Tempforroute = *itTempforward;
			if(Tempforroute.iDestinationid ==
Topmessage.iDestinationid)
			{
				bForwardflag=true;
				break;
			}
		}
		if(bForwardflag==true)
		{
			//Node located fowarding message.
			iPowerlevel = iPowerlevel - 250;

	Returnmessage.Setdatamessage(Topmessage.iSourceid,Topmessage.iDestinationid,Topmessage.iMessageid,iNodeid,Tempforro
ute.iNeighborid);
			return(Returnmessage);
		}
		else
```

```
                                        {
                                                //if the node is missing creating a new RERR message.

    Returnmessage.CreateRERR(iNodeid,Topmessage.iDestinationid,Topmessage.iForwardnodeid);
                                                Returnmessage.iTimetolive = iTime;
                                                Returnmessage.iForwardnodeid = iNodeid;
                                                IMessage_drop++;
                                                return(Returnmessage);
                                        }

                                    }
                                }
                            }
                            else
                            {
                                Returnmessage.iMessagetype = 5; // Ignore message.
                            }

                        }
                        else
                        {
                            Returnmessage.iMessagetype = 6; // No messages to read
                        }
    }
    else
    {
        Returnmessage.iMessagetype = 6; // No messages in list.
    }
    return(Returnmessage);
}

// Default constructor for CAodvNode class.
CAodvNode::CAodvNode(void)
{
    iSequenceno=0;
    iCurrentrreqid =0;
    iNumberofpackets=0;
    iCurrentpacket=0;
    iCurrentdestin=0;
    iCurdesthopcount=0;
    bTransmitdata= false;
    bRouteest= false;
    bReqroute=false;
    iReqcount=0;
    iInitialtime=0;
}
//Copy Constructor of CAodvNode class.
CAodvNode::CAodvNode(const CAodvNode &x1):CNode(x1)
{
  iSequenceno = x1.iSequenceno;
    iCurrentrreqid = x1.iCurrentrreqid;
    iNumberofpackets = x1.iNumberofpackets;
    iCurrentpacket = x1.iCurrentpacket;
    iCurrentdestin =x1.iCurrentdestin;
    iCurdesthopcount= x1.iCurdesthopcount;
    bTransmitdata =x1.bTransmitdata;
    bRouteest = x1.bRouteest;
    bReqroute = x1.bReqroute;
    iReqcount =x1.iReqcount;
    iInitialtime = x1.iInitialtime;
    listReverseroute = x1.listReverseroute;
    listForwardroute =x1.listForwardroute;
    listRoutingtable =x1.listRoutingtable;
    listAdjacency =x1.listAdjacency;
    listMessage =x1.listMessage;
    listGlobal = x1.listGlobal;
}

//= Operator Overloading for CAodvNode class.
CAodvNode & CAodvNode::operator =(CAodvNode x1)
{
    iPowerlevel = x1.iPowerlevel;
```

```
		iSpeed = x1.iSpeed;
		iCurrloc_X = x1.iCurrloc_X;
		iCurrloc_Y = x1.iCurrloc_Y;
		iFutloc_X = x1.iFutloc_X;
		iFutloc_Y = x1.iFutloc_Y;
		iBoundary_X = x1.iBoundary_X;
		iBoundary_Y = x1.iBoundary_Y;
		iMaxspeed = x1.iMaxspeed;
		bMobility = x1.bMobility;
		iBeacontime = x1.iBeacontime;
		iLastbeacon = x1.iLastbeacon;
		iNodeid = x1.iNodeid;
		iPausetime = x1.iPausetime;
		iSequenceno = x1.iSequenceno;
		iCurrentrreqid = x1.iCurrentrreqid;
		iNumberofpackets = x1.iNumberofpackets;
		iCurrentpacket = x1.iCurrentpacket;
		iCurrentdestin =x1.iCurrentdestin;
		iCurdesthopcount= x1.iCurdesthopcount;
		bTransmitdata =x1.bTransmitdata;
		bRouteest = x1.bRouteest;
		bReqroute = x1.bReqroute;
		iReqcount =x1.iReqcount;
		iInitialtime = x1.iInitialtime;
		listReverseroute = x1.listReverseroute;
		listForwardroute =x1.listForwardroute;
		listRoutingtable =x1.listRoutingtable;
		listAdjacency =x1.listAdjacency;
		listMessage =x1.listMessage;
		lMessage_succes = x1.lMessage_succes;
		lMessage_drop = x1.lMessage_drop;
		lOverhead = x1.lOverhead;
		lMessage_sent = x1.lMessage_sent;
		bCounted = x1.bCounted;
		listGlobal = x1.listGlobal;
		return(*this);
}
/*
		The function determines the nodes for which RREQ message
		should be forwarded. The node Ids are maintained in a list
		that is used in upper layer functions to simulate the
		actuall forwarding.
*/
void CAodvNode::Forwardrreq(CAodvmessage Message)
{
		listGlobal.clear();

		list <CAdjacency>::iterator itAdj;
		for(itAdj = listAdjacency.begin(); itAdj != listAdjacency.end(); itAdj++)
		{
				CAdjacency Tempadj = *itAdj;
				if(Tempadj.iNeighborid  != Message.iNodeforwardtoid)
				{
						//Selecting nodes other than the node from which
						//the current node has received RREQ message.
						listGlobal.push_back(Tempadj.iNeighborid);
				}
		}
}
/*
		This function calls the four update table functions.
*/
void CAodvNode::Updatealltables(long iTime)
{
		Update_adjacency(iTime);
		Update_Reverseroute(iTime);
		Update_Forwardroute(iTime);
		Update_Routingtable(iTime);
}

/*
		File Name: AodvSim.h
```

```
		The CAodvSim class declared in this file is used for simulating
		Ad hoc On-demand Distance Vector Routing protocol. It uses the
		CAodvNode class that represents the AODV node.
*/
# pragma once
# include "AodvNode.h"
# include <math.h>
# include <vector>
# include <iostream>
# include <fstream>
using namespace std;
/*
		The CAodvSim class has variables and functions that support the
		simulation of AODV protocol. It has variables to collect the
		simulation results and parameters to specify the simulation
		constraints.
*/
class CAodvSim
{
public:
		vector<CAodvNode> Nodelist; //List of Nodes.
		//Temprorary list of Node IDs
		vector <int> listGlobal1;
		long lClock; //Simulator clock value.
		int iNonodesdied; //Constraint how many node dead.
		int iRadius; //Nodes communication radius.
		int iMaxnodes; //Maximum number of node in network.
		int iMaxdesiredspeed; //Nodes speed
		int iBoundaryX; //X-Coordinate boundary condition.
		int iBoundaryY; //Y-Coordiate boundary condition.
		int iMaxsecurity; //Maximum security available for nodes.
		long iMaxpowerlevel; //Maximum power a node initially has.
		//Simulation results collection.
		long lTotal_message_succesful; //Messages successfully tranmisted.
		long lTotal_overhead; //Overhead messages.
		long lTotal_message_dropped; //Message dropped.
		long lTotal_message_sent; //Total messages sent.
		long lTotal_power; //Power remaining of all the nodes.
		long lTotal_transattempt; //Total transmission attempts.
		long lTotal_message_lost; //Messages lost in network.
		//Functions to support simulation.
		void Send_beacon_to_nodes(int iId);
		CAodvSim(void);
		void Aodvsimulation(ostream &out, unsigned int lSeed, double dPercent);
		void Sendmessage(CAodvmessage Message);
		void Sendrreq(CAodvmessage Message);
		void Aodvsimtime(ostream &out, unsigned int lSeed, long lMaxc);
};

/*
		File Name: AodvSim.cpp
		The file has implementation of all the functions present in
		CAodvSim class.
*/
# include "AodvSim.h"
/*
		The function is used to send beacons between nodes. It has
		a node ID as input. It sends beacons to all the nodes that
		are physically surrounding the input node.
*/
void CAodvSim::Send_beacon_to_nodes(int iId)
{
		unsigned int iTemp;
		//Locating the input node in the list.
		for(iTemp=0; iTemp < Nodelist.size(); iTemp++)
		{
				if(Nodelist[iTemp].iNodeid== iId)
				{
						break;
				}
		}
		//Creating beacon.
```

```
		CAodvmessage Newmessage;
		Newmessage.iSourceid = iId;
		Newmessage.iMessagetype = 0;
		Newmessage.iSourcesequenceno = Nodelist[iTemp].iSequenceno;
		Newmessage.iTimetolive = lClock;
		Newmessage.iForwardnodeid = iId;
		Nodelist[iTemp].iLastbeacon=lClock;
		//Identifying the input node location.
		int xother, yother, xthis, ythis;
  double Caldistance;
  xthis = Nodelist[iTemp].iCurrloc_X;
  ythis = Nodelist[iTemp].iCurrloc_Y;
  unsigned int iTemp1;
  for(iTemp1=0; iTemp1 <Nodelist.size(); iTemp1++)
  {
		if(iTemp != iTemp1)
		{
			//if the node is not the input node.
			//Calculating the distance of the node from input node.
			xother = Nodelist[iTemp1].iCurrloc_X;
			yother = Nodelist[iTemp1].iCurrloc_Y;
			Caldistance=sqrt(double((xother-xthis)*(xother-xthis) + (yother-ythis)*(yother-ythis)));
			if(iRadius >= int(Caldistance))
			{
				//if the node is located with the input node communication radius.
				Newmessage.iDestinationid = iTemp1;
				Newmessage.iNodeforwardtoid = iTemp1;
				Nodelist[iTemp1].listMessage.push_back(Newmessage);
			}
		}
  }
}
/*
		The function sends RREQ message to all the node present
		in the temporary list. The input message is forwarded to a
		node using the Sendmessage function.
*/
void CAodvSim::Sendrreq(CAodvmessage Message)
{
 unsigned int iTemp;
 //For each node present in list.
 for(iTemp=0; iTemp < listGlobal1.size(); iTemp++)
 {
		Message.iNodeforwardtoid = listGlobal1[iTemp];
		Sendmessage(Message);
 }
 listGlobal1.clear();
}
/*
		The function is used to send the input message between
		two nodes. The sender and receiver nodes are identified
		from the message. if the nodes are out of reach then
		the message is ignored.
*/
void CAodvSim::Sendmessage(CAodvmessage Message)
{
 int xother, yother, xthis, ythis;
 double Caldistance;
 int iTonodeid, iSendnodeid;
 //Identifying the sender and recevier.
 iTonodeid = Message.iNodeforwardtoid;
 iSendnodeid = Message.iForwardnodeid;
 unsigned int iTemp;
 //Locating the sender in the node list.
 for(iTemp = 0; iTemp < Nodelist.size(); iTemp ++)
 {
		if(iSendnodeid == Nodelist[iTemp].iNodeid)
		{
			break;
		}
 }
 //Getting the sender node location.
```

```
  xthis = Nodelist[iTemp].iCurrloc_X;
  ythis = Nodelist[iTemp].iCurrloc_Y;
  //Locating the destination in the node list.
 for(iTemp = 0; iTemp < Nodelist.size(); iTemp ++)
 {
		if(iTonodeid == Nodelist[iTemp].iNodeid)
		{
			break;
		}
 }
 //Getting the destination node location.
 xother = Nodelist[iTemp].iCurrloc_X;
 //Calculating the distance between the nodes.
 yother = Nodelist[iTemp].iCurrloc_Y;
 Caldistance=sqrt(double((xother-xthis)*(xother-xthis) + (yother-ythis)*(yother-ythis)));
 if(int(Caldistance) < iRadius)
 {
		//if the nodes are within reach.
		Nodelist[iTemp].listMessage.push_back(Message);
 }
 else
 {
		//if the node are out of reach.
		lTotal_message_lost++;
 }

}
/*
	Default Constructor of the CAodvSim class.
*/
CAodvSim::CAodvSim(void)
{
	lClock=0;
	iNonodesdied=0;
	iRadius=0;
	iMaxnodes=0;
	iMaxdesiredspeed=0;
	iBoundaryX=0;
	iBoundaryY=0;
	iMaxpowerlevel=0;
	iMaxsecurity=0;
	lTotal_message_succesful=0;
	lTotal_overhead=0;
	lTotal_message_dropped=0;
	lTotal_message_sent=0;
	lTotal_power=0;
	lTotal_transattempt=0;
	lTotal_message_lost =0;
}
/*
	The Aodvsimulation function simulates AODV protocol until the
	input percentage of nodes are dead. The function has three inputs
	output file stream, random function seed value and percentage of
	node in network to be dead.
*/
void CAodvSim::Aodvsimulation(ostream &out, unsigned int lSeed, double dPercent)
{
	//Initially assign the desired number of nodes with the given parameters.
	unsigned int iNodes;
	srand(lSeed); //set the random seed value.
	for(iNodes=0; int(iNodes) < iMaxnodes; iNodes++)
	{
		CAodvNode Tempnode;
		Tempnode.iNodeid= iNodes;
		Tempnode.iMaxspeed = iMaxdesiredspeed;
		Tempnode.iBoundary_X = iBoundaryX;
		Tempnode.iBoundary_Y = iBoundaryY;
		Tempnode.iCurrloc_X = rand()%iBoundaryX;
		Tempnode.iCurrloc_Y = rand()%iBoundaryY;
		Tempnode.iPowerlevel = iMaxpowerlevel;
		Tempnode.iSecuritylevel = rand()%iMaxsecurity;
		Nodelist.push_back(Tempnode);
```

```
}
lClock=0; //Initiallize the clock value.

while (iNonodesdied < (iMaxnodes * (dPercent/100)))
{
        //Checks if the number of node dead condition is satisfied.
        lClock++;
        //Each individual node in the network.
        for(iNodes=0; iNodes < Nodelist.size(); iNodes++)
        {
                //The check the power condition.
                if((Nodelist[iNodes].iPowerlevel <=0)&&(Nodelist[iNodes].bCounted == false))
                {
                        //if the node was not previously considered.
                        Nodelist[iNodes].bCounted = true;
                        iNonodesdied++;
                }
                if(Nodelist[iNodes].iPowerlevel >0)
                {
                        //Node has sufficient power resources.
                        if(lClock > 5)
                {
                        //To give time for the network to settle down.
                        Nodelist[iNodes].Set_future();
                        Nodelist[iNodes].Move_node();
                        Nodelist[iNodes].Determine_Send_Messsage(lClock, int(Nodelist.size()));
                        CAodvmessage Tempmessage = Nodelist[iNodes].Send_Message(lClock);
                        if(Tempmessage.iMessagetype != 0)
                        {
                                if(Tempmessage.iMessagetype ==1)
                                {
                                        //Node wants to send RREQ message.
                                        Nodelist[iNodes].Forwardrreq(Tempmessage);
                                        unsigned int iVec;
                                        for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
                                        {
                                                listGlobal1.push_back(Nodelist[iNodes].listGlobal[iVec]);
                                        }
                                        Sendrreq(Tempmessage);
                                }
                                else
                                {
                                        //Send any other data message.
                                        Sendmessage(Tempmessage);
                                }
                        }
                }
                        bool bBeacon;
                        //Check if the node wants to send a beacon.
                        bBeacon=Nodelist[iNodes].Send_beacon(lClock);
                        if(bBeacon== true)
                        {
                                //Send beacon.
                                Send_beacon_to_nodes(iNodes);
                        }
                        CAodvmessage Tempmessage;
                        while(1)
                        {
                                //Read messages of the node.
                        Tempmessage = Nodelist[iNodes].Read_Message(lClock);
                        if(Tempmessage.iMessagetype == 6)
                        {
                                //if they are no message to read at this time
                                break;
                        }
                        if(Tempmessage.iMessagetype == 1)
                        {
                                //Node needs to forward RREQ Message.
                                Nodelist[iNodes].Forwardrreq(Tempmessage);
                                unsigned int iVec;
                                for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
                                {
```

```
                                        listGlobal1.push_back(Nodelist[iNodes].listGlobal[iVec]);
                                    }
                                    Sendrreq(Tempmessage);
                                }
                                //if message type=5 then it ignores it.
                                if(Tempmessage.iMessagetype != 5)
                                {
                                    //Needs to send message.
                                    Sendmessage(Tempmessage);
                                }
                        }
                                Nodelist[iNodes].Updatealltables(lClock);
                        }
                        else
                        {
                                //As no power is left it clears teh messages.
                                Nodelist[iNodes].listMessage.clear();
                        }
                }


        }
        //End of simulation. Collecting the simulation results.
  for(iNodes=0; iNodes < Nodelist.size(); iNodes++)
  {
        lTotal_message_succesful= lTotal_message_succesful + Nodelist[iNodes].lMessage_succes;
        lTotal_overhead= lTotal_overhead + Nodelist[iNodes].lOverhead;
        lTotal_message_dropped= lTotal_message_dropped + Nodelist[iNodes].lMessage_drop;
        lTotal_message_sent= lTotal_message_sent + Nodelist[iNodes].lMessage_sent;
        lTotal_power= lTotal_power + Nodelist[iNodes].iPowerlevel;
        lTotal_transattempt = lTotal_transattempt + Nodelist[iNodes].lTotaltrans;
  }
  //Output the simulation results to a file.
  out<<" Ad hoc On-Demand Distance Vector Routing Protocol Simulation";
  out<<"\n";
  out<<"Number of Nodes: "<<iMaxnodes<<"\n";
  out<<"Simulation Time: "<<lClock<<"\n";
  out<<"Number of Nodes Dead: "<<iNonodesdied<<"\n";
  out<<"Average Power Left of a Node: "<<double(lTotal_power / iMaxnodes)<<"\n";
  out<<"Number of Message Sent: "<<lTotal_message_sent<<"\n";
  out<<"Number of Message reached Destination: "<<lTotal_message_succesful<<"\n";
  out<<"Number of Message dropped: "<<lTotal_message_dropped<<"\n";
  out<<"Total Overhead: "<<lTotal_overhead<<"\n";
  out<<"Total Transmission attempts: "<<lTotal_transattempt<<"\n";
  out<<"Total Messages lost in the Network: "<<lTotal_message_lost<<"\n";
  out<<"% of Messages transmited succesufuly: "<<((double(lTotal_message_succesful)/double(lTotal_message_sent))*100)<<"\n";
  out<<"\n";
}
/*
        The Aodvsimtime funciton is simular to the above function with the
        difference in the constraint. The maximum clock value is the constraint
        overhere. The simulation is performed until that clock value is reached.
*/
void CAodvSim::Aodvsimtime(ostream &out, unsigned int lSeed, long lMaxc)
{
        unsigned int iNodes;
        srand(lSeed);
        //Intialize the node in the network.
        for(iNodes=0; int(iNodes) < iMaxnodes; iNodes++)
        {
                CAodvNode Tempnode;
                Tempnode.iNodeid= iNodes;
                Tempnode.iMaxspeed = iMaxdesiredspeed;
                Tempnode.iBoundary_X = iBoundaryX;
                Tempnode.iBoundary_Y = iBoundaryY;
                Tempnode.iCurrloc_X = rand()%iBoundaryX;
                Tempnode.iCurrloc_Y = rand()%iBoundaryY;
                Tempnode.iPowerlevel = iMaxpowerlevel;
                Tempnode.iSecuritylevel = rand()%iMaxsecurity;
                Nodelist.push_back(Tempnode);
        }
        lClock=0;
```

```
//Check the clock constriant.
while (lClock < lMaxc)
{
	lClock++;
	//For each node in the network.
	for(iNodes=0; iNodes < Nodelist.size(); iNodes++)
	{
		if((Nodelist[iNodes].iPowerlevel <=0)&&(Nodelist[iNodes].bCounted == false))
		{
			Nodelist[iNodes].bCounted = true;
			iNonodesdied++;
		}
		if(Nodelist[iNodes].iPowerlevel >0)
		{
			if(lClock > 5)
		{
			//To give time for the network to settle down.
			Nodelist[iNodes].Set_future();
			Nodelist[iNodes].Move_node();
			Nodelist[iNodes].Determine_Send_Messsage(lClock, int(Nodelist.size()));
			CAodvmessage Tempmessage = Nodelist[iNodes].Send_Message(lClock);
			if(Tempmessage.iMessagetype != 0)
			{
				if(Tempmessage.iMessagetype ==1)
				{
					//Forward the RREQ message.
					Nodelist[iNodes].Forwardrreq(Tempmessage);
					unsigned int iVec;
					for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
					{
						listGlobal1.push_back(Nodelist[iNodes].listGlobal[iVec]);
					}
					Sendrreq(Tempmessage);
				}
				else
				{
					//Send any other message.
					Sendmessage(Tempmessage);
				}
			}
		}
			bool bBeacon;
			//Node needs to send a beacon.
			bBeacon=Nodelist[iNodes].Send_beacon(lClock);
			if(bBeacon== true)
			{
				Send_beacon_to_nodes(iNodes);
			}
			CAodvmessage Tempmessage;
			while(1)
			{
				//Read messages from the list.
			Tempmessage = Nodelist[iNodes].Read_Message(lClock);
			if(Tempmessage.iMessagetype == 6)
			{
				//No more messages to read.
				break;
			}
			if(Tempmessage.iMessagetype == 1)
			{
				//Forward RREQ message.
				Nodelist[iNodes].Forwardrreq(Tempmessage);
				unsigned int iVec;
				for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
				{
					listGlobal1.push_back(Nodelist[iNodes].listGlobal[iVec]);
				}
				Sendrreq(Tempmessage);
			}
			if(Tempmessage.iMessagetype != 5)
			{
				//Forward other messages ignore if message type=5.
```

```
                                        Sendmessage(Tempmessage);
                                }
                        }
                                Nodelist[iNodes].Updatealltables(lClock);
                        }
                        else
                        {
                                Nodelist[iNodes].listMessage.clear();
                        }
                }
        }
        //Collect simulation results.
  for(iNodes=0; iNodes < Nodelist.size(); iNodes++)
  {
        lTotal_message_succesful= lTotal_message_succesful + Nodelist[iNodes].lMessage_succes;
        lTotal_overhead= lTotal_overhead + Nodelist[iNodes].lOverhead;
        lTotal_message_dropped= lTotal_message_dropped + Nodelist[iNodes].lMessage_drop;
        lTotal_message_sent= lTotal_message_sent + Nodelist[iNodes].lMessage_sent;
        lTotal_power= lTotal_power + Nodelist[iNodes].iPowerlevel;
        lTotal_transattempt = lTotal_transattempt + Nodelist[iNodes].lTotaltrans;
  }
  //Output simulation results.
  out<<" Ad hoc On-Demand Distance Vector Routing Protocol Simulation";
  out<<"\n";
  out<<"Number of Nodes: "<<iMaxnodes<<"\n";
  out<<"Simulation Time: "<<lClock<<"\n";
  out<<"Number of Nodes Dead: "<<iNonodesdied<<"\n";
  out<<"Average Power Left of a Node: "<<double(lTotal_power / iMaxnodes)<<"\n";
  out<<"Number of Message Sent: "<<lTotal_message_sent<<"\n";
  out<<"Number of Message reached Destination: "<<lTotal_message_succesful<<"\n";
  out<<"Number of Message dropped: "<<lTotal_message_dropped<<"\n";
  out<<"Total Overhead: "<<lTotal_overhead<<"\n";
  out<<"Total Transmission attempts: "<<lTotal_transattempt<<"\n";
  out<<"Total Messages lost in the Network: "<<lTotal_message_lost<<"\n";
  out<<"% of Messages transmited succesufuly: "<<((double(lTotal_message_succesful)/double(lTotal_message_sent))*100)<<"\n";
  out<<"\n";

}

/*
        File Name: CBRNode.h
        This file has the declaration of all the classes requried
        for supporting Cluster Based Routing (CBR) protocol. The
        implementation of functions present in these classes is
        given in CBRNode.cpp file.
*/
# pragma once
# include "Node.h"

/*
         The CAdajcency_cbr structure is used for maintaining the neighbor
        information for each node. When a node receives a beacon from
        another node then an entry is added or updated in the adjacency
        table. Regular updates are made to have current neighbor information.
*/
class CAdjacency_cbr
{
public:
        int iNeighid; //Neighbor node ID.
        int iClusterid; // Cluster ID.
        int iRole; //Role of Neighbor Node.
        long iTimestamp; //Timestamp.
        //Functions of class.
        CAdjacency_cbr(void);
        CAdjacency_cbr(const CAdjacency_cbr &x1);
        CAdjacency_cbr & operator=(CAdjacency_cbr x1);
};
/*
        The CRreqlist class is used to maintian the list of RREQ
        messages boradcasted by other nodes. When a node receives
        RREQ message it check if an instance of this is already
        present. if not then it adds an entry in the table and
```

```
		forwards the RREQ message to other nodes.
*/
class CRreqlist
{
public:
		int iSourceid; //Source node ID.
		int iRreqid; //RREQ message ID.
		long iTimestamp;//Time stamp.
		//Constructors of class.
		CRreqlist(void);
		CRreqlist(const CRreqlist &x1);
		CRreqlist & operator=(CRreqlist x1);
};
/*
		The CMessage_cbr class has multi-purpose use in this simulation.
   It is used as a message data structure- a medium to send messages
		between the nodes in the network. Each node has a list to store
		all the messages it receives and later reads them. The structure
		is also used as return type variable to exchange parameter values
		between the functions.
*/
class CMessage_cbr
{
public:
		int iMessageid; //Message ID.
		int iMessagetype; //Type of Message.
		int iSenderid; //Sender ID.
		int iSourceid; //Source ID.
		int iReceiverid; //Recevier ID.
		int iDestinationid; //Destination ID.
		long iTimestamp; //Time stamp of message.
		vector <int> listRoute; //Route of the message.
		int iLocroute; //Current node in the route.
		//Functions of the class.
		CMessage_cbr(void);
		CMessage_cbr(const CMessage_cbr &x1);
		CMessage_cbr & operator= (CMessage_cbr x1);
		int Returnrouteloc(void);
		void Addtoroute(int nodeid);
		void Createrreqpacket(int iMessid, int iSour, int iDest);
		void Createbeacon(int iNode, int iClust, int iR, long iTime);
};
/*
		The CCBRNode class represents the Cluster Based Routing
		protocol node in the network. It inherits the CNode class
		and includes other variables and functions to support CBR.
*/
class CCBRNode: public CNode
{
public:
		int iCurstatus; //0-Undecided, 1-Clusterhead, 2-Member, 3-Gateway node.
		int iClusterid; //The cluster it belongs to.
		vector <int> listCurroute; //The current destination route.
		int iCurdestin; //Current destination.
		int iCurpacket; //Current packet transmitted.
		bool bTransmitdata; //if transmitting data.
		bool bRoutereq; //if route requested.
		bool bRouteest; //if route established.
		int iRreqid; // route request ID.
		int iReqcount; //Number of time route requested.
		long iInitialtime; //Time when route requested.
		int iNumberofpackets; //Number of packets to be transmitted.
		vector <int> listGlobal; //Temprorary list.
		vector <int> listClusterheadid; //list of cluster heads if gateway node.
		list <CAdjacency_cbr> listAdjacency; //Adjacency list.
		list <CMessage_cbr> listMessage; //Messages list.
		list <CRreqlist> listRreq; //RREQ list
		//Functions that support CBR protocol.
		bool Checkclusterlist(int iClid);
		void Deleteclusterlist(int iClid);
		bool Determine_Send_Message(long iTime, int iMaxnodes);
		CMessage_cbr Send_Message(long iTime);
```

```
		CMessage_cbr Read_Message(long iTime);
		void Add_adjacency(int iNeig, int iClus, int iR, long iTime);
		void Update_adjacency(long iTime);
		bool Add_Rreqlist(int iSour, int iRid, long iTime);
		void Update_Rreqlist(long iTime);
		bool Check_adjacency(int iNeigh);
		int Decide_cluster(void);
		CCBRNode(void);
		CCBRNode(const CCBRNode &x1);
		CCBRNode & operator=(CCBRNode x1);
		void Forwardrreq(CMessage_cbr Message);
		void Updatealltables(long iTime);

};

/*
		File Name: CBRNode.cpp
		The file has the implementation of functions present in
		the classes belonging to CBRNode.h file.
*/
# include "CBRNode.h"
/*
		Default constructor of CAdjacency_cbr class.
*/
CAdjacency_cbr::CAdjacency_cbr(void)
{
		iNeighid=0;
		iClusterid=0;
		iRole=0;
		iTimestamp=0;
}
/*
		Copy constructor of CAdjacency_cbr class.
*/
CAdjacency_cbr::CAdjacency_cbr(const CAdjacency_cbr &x1)
{
		iNeighid = x1.iNeighid;
		iClusterid = x1.iClusterid;
		iRole = x1.iRole;
		iTimestamp = x1.iTimestamp;
}
/*
		= Operator Overloading for the CAdjacency_cbr class.
*/
CAdjacency_cbr & CAdjacency_cbr::operator =(CAdjacency_cbr x1)
{
		iNeighid = x1.iNeighid;
		iClusterid = x1.iClusterid;
		iRole = x1.iRole;
		iTimestamp = x1.iTimestamp;
		return(*this);
}
/*
		Defualt constructor of the CMessage_cbr class.
*/
CMessage_cbr::CMessage_cbr(void)
{
		iMessageid =0;
		iMessagetype =0;
		iSenderid =0;
		iSourceid =0;
		iReceiverid =0;
		iDestinationid =0;
		iLocroute =0;
		iTimestamp =0;
}
/*
		Copy constructor of the CMessage_cbr class.
*/
CMessage_cbr::CMessage_cbr(const CMessage_cbr &x1)
{
		iMessageid = x1.iMessageid;
```

```
		iMessagetype =x1.iMessagetype;
		iSenderid = x1.iSenderid;
		iSourceid = x1.iSourceid;
		iReceiverid = x1.iReceiverid;
		iDestinationid = x1.iDestinationid;
		iLocroute = x1.iLocroute;
		listRoute = x1.listRoute;
		iTimestamp = x1.iTimestamp;
}
/*
		= Operator overloading for the CMessage_cbr class.
*/
CMessage_cbr& CMessage_cbr::operator =(CMessage_cbr x1)
{
		iMessageid = x1.iMessageid;
		iMessagetype =x1.iMessagetype;
		iSenderid = x1.iSenderid;
		iSourceid = x1.iSourceid;
		iReceiverid = x1.iReceiverid;
		iDestinationid = x1.iDestinationid;
		iLocroute = x1.iLocroute;
		listRoute = x1.listRoute;
		iTimestamp = x1.iTimestamp;
		return(*this);
}
/*
		The function returns a node id that is being refered by
		iLocroute variable in the listRoute list present in
		CMessage_cbr class. if the reference is invalid then
		it returns -1.
*/
int CMessage_cbr::Returnrouteloc(void)
{
		if(iLocroute <0)
		{
				return(-1);
		}
		if(iLocroute >= int(listRoute.size()))
		{
				return(-1);
		}
		return(listRoute[iLocroute]);
}
/*
		The function adds a node id to the current list route.
*/
void CMessage_cbr::Addtoroute(int nodeid)
{
		listRoute.push_back(nodeid);
}
/*
		The function is used to create a RREQ packet, it takes the
		message id, source id, and destination id as input.
*/
void CMessage_cbr::Createrreqpacket(int iMessid, int iSour, int iDest)
{
		iMessagetype =1;
		iMessageid = iMessid;
		iSourceid = iSour;
		iDestinationid = iDest;
		listRoute.clear(); //Clears the list.
		listRoute.push_back(iSour);
}
/*
		The function creates a beacon, it take the Node id,
		cluster ID, message ID, and current clock value as input.
*/
void CMessage_cbr::Createbeacon(int iNode, int iClust, int iR, long iTime)
{
		iMessagetype =0;
		iSourceid = iNode;
		iSenderid = iClust;
```

```
		iMessageid = iR;
		iTimestamp = iTime;
}
//Constructor for CRreqlist class.
CRreqlist::CRreqlist(void)
{
		iSourceid =0;
		iRreqid =0;
		iTimestamp=0;
}
//Copy Constructor of the CRreqlist class.
CRreqlist::CRreqlist(const CRreqlist &x1)
{
		iSourceid = x1.iSourceid;
		iRreqid = x1.iRreqid;
		iTimestamp = x1.iTimestamp;
}
//= Operator Overloading of the CRreqlist class.
CRreqlist& CRreqlist::operator =(CRreqlist x1)
{
		iSourceid = x1.iSourceid;
		iRreqid = x1.iRreqid;
		iTimestamp = x1.iTimestamp;
		return(*this);
}
/*
		The function is used to check if the input cluster id is
		already present  in the Clusterhead list. Returns true if
		it is present.
*/
bool CCBRNode::Checkclusterlist(int iClid)
{
		bool bCheckflag;
		bCheckflag = false;
		unsigned int i;
		for(i=0;i <listClusterheadid.size(); i++)
		{
				if(listClusterheadid[i]==iClid)
				{
						bCheckflag=true;
						break;
				}
		}
		return(bCheckflag);
}
/*
		The function is used to delete the input cluster Id from
		the Cluster head list.
*/
void CCBRNode::Deleteclusterlist(int iClid)
{
		vector <int> Tempvector;
		unsigned int iIt;
		//Move the current list to temp list.
		for(iIt=0; iIt <listClusterheadid.size(); iIt++)
		{
    if(listClusterheadid[iIt]!= iClid)
		  {
				  //ignore if input cluster ID.
				  Tempvector.push_back(listClusterheadid[iIt]);
		  }
		}
		listClusterheadid.clear();
		//Copy back to Cluster head list.
		for(iIt=0; iIt <Tempvector.size(); iIt++)
		{
				listClusterheadid.push_back(Tempvector[iIt]);
		}

}
/*
		The function is used to add an entry in the adjacency list. if
```

```
		the entry is already present then the timestamp is updated. The
		neighbor id, cluster id, role of node, and timestamp are given
		as input.
*/
void CCBRNode::Add_adjacency(int iNeig, int iClus, int iR, long iTime)
{
		list <CAdjacency_cbr>::iterator itPointer;
		//Locate the node in adjacency list.
		for(itPointer= listAdjacency.begin(); itPointer!= listAdjacency.end(); itPointer++)
		{
			CAdjacency_cbr Tempadjacency =*itPointer;
			if(Tempadjacency.iNeighid == iNeig)
			{
				listAdjacency.erase(itPointer);
				break;
			}
		}
		CAdjacency_cbr Tempadj;
		Tempadj.iClusterid = iClus;
		Tempadj.iNeighid = iNeig;
		Tempadj.iRole = iR;
		Tempadj.iTimestamp = iTime;
		listAdjacency.push_back(Tempadj);
}
/*
		The function is used to maintain updated values in the table.
		Any entry in the table with expired timestamp is removed.
*/
void CCBRNode::Update_adjacency(long iTime)
{
		list <CAdjacency_cbr>::iterator itPointer;
		list <CAdjacency_cbr> listTemp;
		int iT;
		if(bMobility == true)
		{
			//if the node is mobile.
			iT= 10;
		}
		else
		{
			//if node is stationary.
			iT = 15;
		}
		//Move nodes to temp list.
		while(!listAdjacency.empty())
		{
			itPointer= listAdjacency.begin();
			CAdjacency_cbr Curr_node = *itPointer;
			if((iTime-Curr_node.iTimestamp)<=iT)
			{
				//if time stamp is not expired.
				listTemp.push_back(Curr_node);
			}
			listAdjacency.pop_front();
		}
		//Move back to the adjacency list.
		while(!listTemp.empty())
		{
			itPointer = listTemp.begin();
			CAdjacency_cbr Curr_node = *itPointer;
	listAdjacency.push_back(Curr_node);
			listTemp.pop_front();
		}
}

/*
		The function adds a RREQ entry in the RREQ list. It has the
		Source id, RREQ id, and timestamp as input. if an entry was
		previously present then it ignores it and does'nt forward the
		RREQ message to other nodes.
*/
bool CCBRNode::Add_Rreqlist(int iSour, int iRid, long iTime)
```

```
{
		list <CReqlist>::iterator itPointer;
		bool bFlag;
		bFlag=true;
		//Locate the RREQ entry if present.
		for(itPointer = listRreq.begin(); itPointer != listRreq.end(); itPointer++)
		{
				CReqlist Tempnode=*itPointer;
				if((Tempnode.iRreqid ==iRid)&&(Tempnode.iSourceid == iSour))
				{
						bFlag=false;
						break;
				}
		}
		if(bFlag==true)
		{
				//Add the RREQ entry into list.
				CReqlist Addnode;
				Addnode.iRreqid = iRid;
				Addnode.iSourceid = iSour;
				Addnode.iTimestamp = iTime;
				listRreq.push_back(Addnode);
		}
		return(bFlag);
}
/*
		The function is used to maintain updated values in the table.
		Any entry in the table with expired timestamp is removed.
*/
void CCBRNode::Update_Rreqlist(long iTime)
{
		list <CReqlist>::iterator itPointer;
		list <CReqlist> listTemp;
		//Move to temp list.
		while(!listRreq.empty())
		{
				itPointer= listRreq.begin();
				CReqlist Curr_node = *itPointer;
				if((iTime-Curr_node.iTimestamp)<=20)
				{
						//Ignore is time expired.
						listTemp.push_back(Curr_node);
				}
				listRreq.pop_front();
		}
		//Copy from temp list to Rreq list.
		while(!listTemp.empty())
		{
				itPointer = listTemp.begin();
				CReqlist Curr_node = *itPointer;
	listRreq.push_back(Curr_node);
				listTemp.pop_front();
		}
}
/*
		The function Determine_Send_Message decides if a node
		wants to transmit data to another node. it has the current
		clock value and maximum number of nodes present as inputs.
		if a node randomly decides to transmit data then the
		function returns true else false. The function also decides
		the destination node and the number of packets to be
		transmitted.
*/
bool CCBRNode::Determine_Send_Message(long iTime, int iMaxnodes)
{
		if(bTransmitdata == false)
		{
		int iProb, iTempnode, iMul;
		iProb = rand()%10;
	if(iProb < 8)
		{
				//When no data is to be sent
```

```
                    return(false);
            }
            else
            {
                    //When data is to be sent.
    iTempnode = rand()%iMaxnodes; //Determine the destination id.
             if(iTempnode == iNodeid)
             {
                    if(iTempnode < (iMaxnodes-2))
                    {
                            iTempnode++;
                    }
                    else
                    {
                            if(iTempnode == 0)
                            {
                                    iTempnode++;
                            }
                            else
                            {
                                    iTempnode--;
                            }
                    }
             }
             //Reset all the values.
             iCurdestin = iTempnode;
             iMul = rand()%10;
             iMul++;
             iNumberofpackets = 100* iMul;
             bRouteest = false;
             bRoutereq=false;
             bTransmitdata = true;
             iCurpacket =0;
             lTotaltrans++;
             return(true);
   }
            }
            return(false);
}
/*
            The function Send_Message decides the packet to be sent. it has
            the current simulator clock value as the input. It returns a
            CAodvmessage that is read by network layer functions. The function
            decides whether to broadcast a RREQ packet. Once the route is
            established it transmits the packets one after the other. The
            function send a maximum of 3 RREQ packets for each transmission.
            if all three RREQ packets are timed out then transmission is
            aborted.
*/
CMessage_cbr CCBRNode::Send_Message(long iTime)
{
            CMessage_cbr Returnmessage;
            if(bTransmitdata == true)
            {
                    //if the node is transmitting data.
            Returnmessage.iDestinationid = iCurdestin;
            Returnmessage.iSourceid = iNodeid;
            Returnmessage.iSenderid = iNodeid;
  Returnmessage.iTimestamp = iTime;
            if(bRouteest==false)
            {
                    //if the Route was not established.
                    if(bRoutereq==false)
                    {
                            //if the route ws not requested before.
                            list <CAdjacency_cbr>::iterator itPointer;
                            for(itPointer = listAdjacency.begin(); itPointer != listAdjacency.end(); itPointer++)
                            {
                                    CAdjacency_cbr Currnode=*itPointer;
                                    if(Currnode.iNeighid == iCurdestin)
                                    {
                                            //The destination is one of the adjacent neighbors.
```

```
                        bRouteest=true;
                        listCurroute.push_back(iNodeid);
                        listCurroute.push_back(Currnode.iNeighid);
                        Returnmessage.iMessageid = iCurpacket;
                        Returnmessage.iMessagetype = 4; //For data message.
                        Returnmessage.iLocroute =1;
                        Returnmessage.Addtoroute(iNodeid);
                        Returnmessage.Addtoroute(Currnode.iNeighid);
                        Returnmessage.iSenderid = iNodeid;
                        Returnmessage.iReceiverid = Currnode.iNeighid;
                        Returnmessage.iTimestamp = iTime;
                        iPowerlevel = iPowerlevel - 250;
                        lMessage_sent++;
                        return(Returnmessage);
                    }
                }

                //if Destination is not present in the adjacency table.
                //A Route request is being sent.
                Returnmessage.Createrreqpacket(iRreqid, iNodeid, iCurdestin);
                Returnmessage.iReceiverid = -1;
                iPowerlevel = iPowerlevel -75;
                Returnmessage.iTimestamp = iTime;
                bRoutereq=true;
                iInitialtime= iTime;
                lOverhead++;
                iRreqid++;//Increment the Route reqeust id.
                iReqcount=0;
                return(Returnmessage);
            }
            else
            {
                if((iTime- iInitialtime)>10)
                {
                    if(iReqcount <3)
                    {
                    //Send request again.
                    Returnmessage.Createrreqpacket(iRreqid, iNodeid, iCurdestin);
                    Returnmessage.iReceiverid = -1;
                    iPowerlevel = iPowerlevel - 75;
                    Returnmessage.iTimestamp = iTime;
                    lOverhead++;
                    iRreqid++;
                    iReqcount++;
                    iInitialtime = iTime;
                    return(Returnmessage);
                    }
                    else
                    {
                        //Cancel the data transmission request.
                        bTransmitdata =false;
                        bRouteest=false;
                        bRoutereq=false;
                        Returnmessage.iMessagetype =0; //Do nothing.
                        return(Returnmessage);
                    }
                }
                else
                {
                    Returnmessage.iMessagetype =0; //Do nothing.
                    return(Returnmessage);
                }
            }
        }
        else
        {
            // when the route is already established.
            if(iCurpacket < iNumberofpackets)
            {
                //Send a Data packet.
                iCurpacket++;
                Returnmessage.iSourceid = iNodeid;
```

```
					Returnmessage.iDestinationid = iCurdestin;
					Returnmessage.iLocroute =1;
					Returnmessage.iSenderid = iNodeid;
					Returnmessage.listRoute = listCurroute;
					Returnmessage.iReceiverid = Returnmessage.Returnrouteloc();
					if(Returnmessage.iReceiverid == -1)
					{
						bTransmitdata =false;
						bRouteest =false;
						bRoutereq =false;
						Returnmessage.iMessagetype =0; //Do nothing.
						return(Returnmessage);
					}
					Returnmessage.iMessageid = iCurpacket;
					Returnmessage.iMessagetype =4;
					Returnmessage.iTimestamp = iTime;
					iPowerlevel = iPowerlevel - 250;
					lMessage_sent++;
					return(Returnmessage);
				}
				else
				{
					//When all the packets are transmitted.
					bTransmitdata =false;
					bRouteest =false;
					bRoutereq =false;
					Returnmessage.iMessagetype =0; //Do nothing.
					return(Returnmessage);
				}

		}
		}
		Returnmessage.iMessagetype =0; //Do nothing.
		return(Returnmessage);
}
/*
		The function is used to check if a neighbor node is
		present in the adjacency list.
*/
bool CCBRNode::Check_adjacency(int iNeigh)
{
		bool bFlag;
		bFlag=false;
		list <CAdjacency_cbr>::iterator itPointer;
		//Check in the adjacency list.
		for(itPointer= listAdjacency.begin(); itPointer!= listAdjacency.end(); itPointer++)
		{
				CAdjacency_cbr Tempnode= *itPointer;
				if(Tempnode.iNeighid == iNeigh)
				{
					//if node id matched.
					bFlag=true;
					break;
				}
		}
		return(bFlag);
}
/*
		This function is one of the basic repsonsibilities of a node.
		Function is used to read the messages it has received. The messages
		are present in a queue and it reads them one at a time. After reading
		a message the node has to perform an action appropriately. The
		function is called until the list is empty or message time stamp is
		less than the current clock time.
*/
CMessage_cbr CCBRNode::Read_Message(long iTime)
{
		CMessage_cbr Returnmessage;
		Returnmessage.iTimestamp = iTime;
		Returnmessage.iSenderid = iNodeid;
		if(!listMessage.empty())
		{
```

```
//if list is not empty.
 if(iPowerlevel <=0)
 {
		//if no power is left.
		Returnmessage.iMessagetype = 6; //Node is dead.
		listMessage.clear();
		return(Returnmessage);
 }
CMessage_cbr Topmessage= listMessage.front();
if(Topmessage.iTimestamp < iTime)
{
	//The message is to be read.
	listMessage.pop_front();
	if((iTime - Topmessage.iTimestamp) > 10)
	{
		//Message timeout.
		iPowerlevel = iPowerlevel -5;
		Returnmessage.iMessagetype =5; //Ignore message timeout.
		return(Returnmessage);
	}
	if(Topmessage.iReceiverid == iNodeid)
	{
		//if the message is received to the correct node.
		if(Topmessage.iMessagetype ==0)
		{
			//Received a Beacon from a neighbor node.
			iPowerlevel = iPowerlevel - 5;
if(iCurstatus==0)
			{
				// if the node still is undecided.
				if(Topmessage.iSourceid < iNodeid)
				{
					//if the neighbor has smaller id.
					iCurstatus=2;
					iClusterid= Topmessage.iSourceid;
					listClusterheadid.push_back(Topmessage.iSourceid);

Add_adjacency(Topmessage.iSourceid,Topmessage.iSenderid,Topmessage.iMessageid, iTime);
					Returnmessage.iMessagetype=5; //Do nothing message.
					return(Returnmessage);
				}
				else
				{
					//if the node has smaller id.
					iCurstatus=1; //Node is the cluster head.
					iClusterid = iNodeid;
					listClusterheadid.push_back(iNodeid);

Add_adjacency(Topmessage.iSourceid,Topmessage.iSenderid,Topmessage.iMessageid,iTime);
					Returnmessage.iMessagetype=5; //Do nothing message.
					return(Returnmessage);
				}
			}
			if(iCurstatus==1)
			{
				//if the node is a clusterhead.
				if(Topmessage.iSourceid < iNodeid)
				{
					//When the neighbor has lower id.
					if(Topmessage.iMessageid ==3)
					{
						//if the node was a gateway node.
						Add_adjacency(Topmessage.iSourceid,
Topmessage.iSenderid, Topmessage.iMessageid, iTime);

						Returnmessage.iMessagetype =5; //Do
nothing message.

						return(Returnmessage);
					}
					else
					{
						//if the node was not a gateway node.
						if(Topmessage.iMessageid == 2)
```

```
                                                {
                                                    //if the node was a member node.

        Add_adjacency(Topmessage.iSourceid, Topmessage.iSenderid, Topmessage.iMessageid, iTime);
                                                    Returnmessage.iMessagetype =5;
//Do nothing message.
                                                    return(Returnmessage);
                                                }
                                                else
                                                {
                                                    //if the node is a clusterhead or
undecided node
                                                    iClusterid =Topmessage.iSourceid;
                                                    iCurstatus = 2;
                                                    listClusterheadid.clear();

        listClusterheadid.push_back(Topmessage.iSourceid);

        Add_adjacency(Topmessage.iSourceid, Topmessage.iSenderid, Topmessage.iMessageid, iTime);
                                                    Returnmessage.iMessagetype =5;
// Do nothing.
                                                    return(Returnmessage);
            }
                                            }
                                        }
                                        else
                                        {
                                            Add_adjacency(Topmessage.iSourceid,
Topmessage.iSenderid, Topmessage.iMessageid, iTime);
                                            Returnmessage.iMessagetype =5; //Do nothing.
                                            return(Returnmessage);
                                        }
        }
                                if(iCurstatus==2)
                                {
                                    // if the node is a member.
                                    Add_adjacency(Topmessage.iSourceid, Topmessage.iSenderid,
Topmessage.iMessageid, iTime);
                                    Returnmessage.iMessagetype =5; // Do nothing.
                                    if(Topmessage.iMessageid ==1)
                                    {
                                        if(Checkclusterlist(Topmessage.iSourceid)==false)
                                        {
                                        listClusterheadid.push_back(Topmessage.iSourceid);
                                        if(listClusterheadid.size() >=2)
                                        {
                                            iCurstatus=3;
                                            iClusterid = listClusterheadid[0];
                                        }
                                        }
                                    }
                                    if(Topmessage.iMessageid >=2)
                                    {
                                        if(Checkclusterlist(Topmessage.iSourceid)==true)
                                        {
                                          Deleteclusterlist(Topmessage.iSourceid);
                                            if(listClusterheadid.size()==0)
                                            {
                                                iCurstatus=1;
                                            }
                                        }
                                    }
                                    return(Returnmessage);
                                }
                                if(iCurstatus==3)
                                {
                                    //if the node is a Gateway node.
                                    Add_adjacency(Topmessage.iSourceid, Topmessage.iSenderid,
Topmessage.iMessageid, iTime);
                                    Returnmessage.iMessagetype =5; // Do nothing.
                                    if(Topmessage.iMessageid ==1)
                                    {
```

```
                                                if(Checkclusterlist(Topmessage.iSourceid)==false)
                                                {
                listClusterheadid.push_back(Topmessage.iSourceid);
                                                }
                                        }
                                        if(Topmessage.iMessageid >=2)
                                        {
                                                if(Checkclusterlist(Topmessage.iSourceid)==true)
                                                {
                                                        Deleteclusterlist(Topmessage.iSourceid);
                                                        if(listClusterheadid.size() ==1)
                                                        {
                                                                iCurstatus=2;
                                                        }
                                                }
                                        }
                                        return(Returnmessage);
                                }
                        }
                        if(Topmessage.iMessagetype ==1)
                        {
                                //Received a Route Request from a node.
                                iPowerlevel = iPowerlevel - 50;

                                if(Add_Rreqlist(Topmessage.iSourceid, Topmessage.iMessageid, iTime)==true)
                                {
                                        //if the request was not previously received.
                                        if(Topmessage.iDestinationid == iNodeid)
                                        {
                                                //if the node is the destination.
                                                Returnmessage.iMessagetype = 2; //Sending a route
reply.
                                                Returnmessage.iSourceid = iNodeid;
                                                Returnmessage.iDestinationid = Topmessage.iSourceid;
                                                Returnmessage.iLocroute =1;
                                                Returnmessage.listRoute.clear();
                                                Returnmessage.listRoute.push_back(iNodeid);
                                                while(!Topmessage.listRoute.empty())
                                                {
                                                        int iTemp, iSize;
                                                        iSize = int(Topmessage.listRoute.size());
                                                        iTemp = Topmessage.listRoute[(iSize-1)];
                                                        Topmessage.listRoute.pop_back();
                                                        Returnmessage.listRoute.push_back(iTemp);
                                                }
                                                Returnmessage.iSenderid = iNodeid;
                                                Returnmessage.iReceiverid =
Returnmessage.Returnrouteloc();
                                                if(Returnmessage.iReceiverid == -1)
                                                {
                                                        Returnmessage.iMessagetype =5; //Do
nothing.
                                                        return(Returnmessage);
                                                }
                                                return(Returnmessage);
                                        }
                                        else
                                        {
                                                //if the node is an intermediate node.
                                                Returnmessage.iMessagetype =1; //continuing rreq.
                                                Returnmessage.iSourceid = Topmessage.iSourceid;
                                                Returnmessage.iDestinationid =
Topmessage.iDestinationid;
                                                Returnmessage.iMessageid = Topmessage.iMessageid;
                                                Returnmessage.iSenderid = iNodeid;
                                                Returnmessage.listRoute = Topmessage.listRoute;
                                                Returnmessage.listRoute.push_back(iNodeid);
                                                iPowerlevel = iPowerlevel - 75;
                                                Returnmessage.iTimestamp = Topmessage.iTimestamp;
                                                Returnmessage.iReceiverid = Topmessage.iSenderid;
//Do not forward to this node.
```

```
					//The receiver will be decided at the next level.
					lOverhead++;
					return(Returnmessage);
				}
			}
			else
			{
				Returnmessage.iMessagetype= 5; //Do nothing.
				return(Returnmessage);
			}
		}
		if(Topmessage.iMessagetype ==2)
		{
			//Received a Route Reply Message.
			if((Topmessage.iDestinationid == iNodeid)&&(bRouteest==false))
			{
				bRouteest=true;
				iCurpacket=0;
				listCurroute.clear();
				listCurroute.push_back(iNodeid);
				while(!Topmessage.listRoute.empty())
				{
					int iTemp, iSize;
					iSize = int(Topmessage.listRoute.size());
					iTemp = Topmessage.listRoute[(iSize-1)];
					Topmessage.listRoute.pop_back();
					listCurroute.push_back(iTemp);
				}
				Returnmessage.iMessagetype = 5; //Do nothing message.
				return(Returnmessage);
			}
			else
			{
				//An intermediate node in the Route reply.
				Returnmessage.iMessagetype =2;
				Returnmessage.iSourceid = Topmessage.iSourceid;
				Returnmessage.iDestinationid = Topmessage.iDestinationid;
				Returnmessage.iSenderid = iNodeid;
				Returnmessage.iLocroute = Topmessage.iLocroute+1;
				Returnmessage.listRoute = Topmessage.listRoute;
				Returnmessage.iReceiverid = Returnmessage.Returnrouteloc();
				if(Returnmessage.iReceiverid == -1)
				{
					Returnmessage.iMessagetype =5; //Do nothing.
					return(Returnmessage);
				}
				return(Returnmessage);
			}
		}
		if(Topmessage.iMessagetype == 3)
		{
			// Recevied a Route Error Message.
			if((Topmessage.iDestinationid == iCurdestin)&&(Topmessage.iSourceid ==
iNodeid))
			{
				bTransmitdata = false;
				bRouteest = false;
				bRoutereq = false;
				Returnmessage.iMessagetype = 5; //Do nothing.
				return(Returnmessage);
			}
			else
			{
				//Forward the RRER message.
				Returnmessage.iMessagetype =3;
				Returnmessage.iDestinationid = Topmessage.iDestinationid;
				Returnmessage.iSourceid = Topmessage.iSourceid;
				Returnmessage.iSenderid = iNodeid;
				Returnmessage.iLocroute = Topmessage.iLocroute+1;
				Returnmessage.listRoute = Topmessage.listRoute;
				Returnmessage.iReceiverid = Returnmessage.Returnrouteloc();
				if(Returnmessage.iReceiverid == -1)
```

```
                        {
                            Returnmessage.iMessagetype = 5; //Do nothing.
                            return(Returnmessage);
                        }
                        return(Returnmessage);
                    }
                }
    if(Topmessage.iMessagetype ==4)
                {
                    //Recevied a Data packet.
                    iPowerlevel = iPowerlevel - 150;
                    if(Topmessage.iDestinationid == iNodeid)
                    {
                        //Packet reached destination.
                        Returnmessage.iMessagetype =5; //Do nothing.
                        lMessage_succes++;
                        return(Returnmessage);
                    }
                    else
                    {   //Packet forwarded by intermediate node.
                        Returnmessage.iMessagetype =4;

                        Returnmessage.iSenderid = iNodeid;
                        Returnmessage.iLocroute = Topmessage.iLocroute +1;
                        Returnmessage.listRoute = Topmessage.listRoute;
                        Returnmessage.iReceiverid = Returnmessage.Returnrouteloc();
                        if(Returnmessage.iReceiverid == -1)
                        {
                            lMessage_drop++;
                            Returnmessage.iMessagetype =5; //Do nothing.
                            return(Returnmessage);
                        }
                        if(Check_adjacency(Returnmessage.iReceiverid)==false)
                        {
                            //if message is not present in adjacency list.
                            lMessage_drop++;
                            Returnmessage.listRoute.clear();
                            int iLoc;
                            for(iLoc =(Returnmessage.iLocroute -1); iLoc >=0; iLoc-
-)
                            {

Returnmessage.listRoute.push_back(Topmessage.listRoute[iLoc]);
                            }
                            Returnmessage.iLocroute =1;
                            Returnmessage.iReceiverid =
Returnmessage.Returnrouteloc();
                            if(Returnmessage.iReceiverid == -1)
                            {
                                Returnmessage.iMessagetype = 5; //Do
nothing.
                                return(Returnmessage);
                            }
                            Returnmessage.iMessagetype =3; //RERR Message.
                            Returnmessage.iDestinationid =
Topmessage.iDestinationid;
                            Returnmessage.iSourceid = Topmessage.iSourceid;
                        }
                        else
                        {
                            //Forward message.
                            iPowerlevel = iPowerlevel - 250;
                            Returnmessage.iMessageid = Topmessage.iMessageid;
                            Returnmessage.iDestinationid =
Topmessage.iDestinationid;
                            Returnmessage.iSourceid = Topmessage.iSourceid;
                        }
                        return(Returnmessage);
                    }
                }
                Returnmessage.iMessagetype =5; //Do nothing.
                return(Returnmessage);
```

```
				}
				else
				{
						//The message is received to the wrong node.
						Returnmessage.iMessagetype =5; //Do nothing message.
						return(Returnmessage);
				}

			}
			else
			{
				//No more messages to be read.
				Returnmessage.iMessagetype =6;
				return(Returnmessage);
			}
		}
		else
		{
			Returnmessage.iMessagetype = 6; //  No more messages to read.
			return(Returnmessage);
		}
		Returnmessage.iMessagetype =6;
		return(Returnmessage);
}
/*
		The function is used to make sure that the node acts according
		to the CBR protocol. if the cluster head node has a new node
		having smaller ID then the node should no longer act as cluster
		head. similar to this other rule other rules of CBR are protected
		in this function.
*/
int CCBRNode::Decide_cluster(void)
{
		if(iCurstatus==1)
		{
			//if the node is currently a clusterhead.
			bool bFlag;
			bFlag = false;
			list <CAdjacency_cbr>::iterator itPointer;
			//Check the adjacency of the node.
			for(itPointer= listAdjacency.begin(); itPointer != listAdjacency.end(); itPointer++)
			{
				CAdjacency_cbr Tempnode=*itPointer;
				if(bFlag==false)
				{
					if((Tempnode.iNeighid < iNodeid) && (Tempnode.iRole !=3))
					{
						//if a node in adjacency has lower id and is not a gateway node.
						iCurstatus =2;
						iClusterid = Tempnode.iNeighid;
						listClusterheadid.clear();
						listClusterheadid.push_back(Tempnode.iNeighid);
						bFlag=true;
					}
				}
				else
				{
					if((Tempnode.iNeighid < iClusterid)&&(Tempnode.iRole ==1))
					{
						iCurstatus=3; //make the node a gatewaynode.
						listClusterheadid.push_back(Tempnode.iNeighid);
					}
				}
			}
			return(0);
		}
		if(iCurstatus ==2)
		{
			//if the node is currently a Member.
			if(Check_adjacency(iClusterid)==false)
			{
				//The current cluster head is missing.
```

```
					listClusterheadid.clear();
					list <CAdjacency_cbr>::iterator itPointer;
					for(itPointer= listAdjacency.begin(); itPointer != listAdjacency.end(); itPointer++)
					{
						CAdjacency_cbr Tempnode=*itPointer;
						if(Tempnode.iRole ==1)
						{
							//if a new cluster head is seen in the adjacency.
							iClusterid = Tempnode.iNeighid;
							listClusterheadid.push_back(Tempnode.iNeighid);
							break;
						}
					}
					if(listClusterheadid.size()==0)
					{
						iCurstatus =1;
					}
					if(listClusterheadid.size() >2)
					{
						iCurstatus =3;
					}
				}
				return(0);
		}
		if(iCurstatus==3)
		{
				//if the node is a gateway node.
				switch(listClusterheadid.size())
				{
				case 0:
						{
							//Cluster head.
							iCurstatus=1;
							break;
						}
				case 1: {
							//Member node.
							iCurstatus = 2;
							iClusterid = listClusterheadid[0];
							break;
						}
				default: break;
				}
				return(0);
		}
		return(0);
}

//Default constructor of the CCBRNode class.
CCBRNode::CCBRNode(void)
{
		iCurstatus=0;
		iClusterid=0;
		iCurdestin=0;
		iCurpacket=0;
		bTransmitdata=false;
		bRoutereq=false;
		bRouteest=false;
		iRreqid=0;
		iReqcount=0;
		iInitialtime=0;
		iNumberofpackets=0;
}
//Copy constructor of the CCBRNode class.
CCBRNode::CCBRNode(const CCBRNode &x1):CNode(x1)
{
		iCurstatus=x1.iCurstatus;
		iClusterid= x1.iClusterid;
		iCurdestin= x1.iCurdestin;
		iCurpacket= x1.iCurpacket;
		bTransmitdata= x1.bTransmitdata;
		bRoutereq=x1.bRoutereq;
```

```
		bRouteest=x1.bRouteest;
		iRreqid= x1.iRreqid;
		iReqcount= x1.iReqcount;
		iInitialtime= x1.iInitialtime;
		iNumberofpackets= x1.iNumberofpackets;
		listGlobal = x1.listGlobal;
		listClusterheadid = x1.listClusterheadid;
		listAdjacency = x1.listAdjacency;
		listMessage = x1.listMessage;
		listRreq = x1.listRreq;
		listCurroute = x1.listCurroute;
}
// = Operator Overloading for the CCBRNode class.
CCBRNode & CCBRNode::operator =(CCBRNode x1)
{
		iPowerlevel = x1.iPowerlevel;
		iSecuritylevel = x1.iSecuritylevel;
		iSpeed = x1.iSpeed;
		iCurrloc_X = x1.iCurrloc_X;
		iCurrloc_Y = x1.iCurrloc_Y;
		iFutloc_X = x1.iFutloc_X;
		iFutloc_Y = x1.iFutloc_Y;
		iBoundary_X = x1.iBoundary_X;
		iBoundary_Y = x1.iBoundary_Y;
		iMaxspeed = x1.iMaxspeed;
		bMobility = x1.bMobility;
		iBeacontime = x1.iBeacontime;
		iLastbeacon = x1.iLastbeacon;
		iNodeid = x1.iNodeid;
		iPausetime = x1.iPausetime;
		lMessage_succes = x1.lMessage_succes;
		lMessage_drop = x1.lMessage_drop;
		lOverhead = x1.lOverhead;
		lMessage_sent = x1.lMessage_sent;
		bCounted = x1.bCounted;
		iCurstatus=x1.iCurstatus;
		iClusterid= x1.iClusterid;
		iCurdestin= x1.iCurdestin;
		iCurpacket= x1.iCurpacket;
		bTransmitdata= x1.bTransmitdata;
		bRoutereq=x1.bRoutereq;
		bRouteest=x1.bRouteest;
		iRreqid= x1.iRreqid;
		iReqcount= x1.iReqcount;
		iInitialtime= x1.iInitialtime;
		iNumberofpackets= x1.iNumberofpackets;
		listGlobal = x1.listGlobal;
		listClusterheadid = x1.listClusterheadid;
		listAdjacency = x1.listAdjacency;
		listMessage = x1.listMessage;
		listRreq = x1.listRreq;
		listCurroute = x1.listCurroute;
		return(*this);
}

/*
		The function determines the nodes for which RREQ message
		should be forwarded. The node Ids are maintained in a list
		that is used in upper layer functions to simulate the
		actuall forwarding.
*/
void CCBRNode::Forwardrreq(CMessage_cbr Message)
{
		listGlobal.clear();
		list <CAdjacency_cbr>::iterator itAdj;

   if(Check_adjacency(Message.iDestinationid)==true)
		{
				listGlobal.push_back(Message.iDestinationid);
		}
		else
		{
```

```
			for(itAdj = listAdjacency.begin(); itAdj != listAdjacency.end(); itAdj++)
			{
				CAdjacency_cbr Tempadj = *itAdj;
				if((Tempadj.iNeighid  != Message.iReceiverid )&&((Tempadj.iRole==1)||(Tempadj.iRole==3)))
				{
					//Forward to cluster heads or gateway nodes.
					listGlobal.push_back(Tempadj.iNeighid);
				}
			}
		}
}

/*
		Updates the tables by calling the respective functions.
*/
void CCBRNode::Updatealltables(long iTime)
{
		Update_adjacency(iTime);
		Update_Rreqlist(iTime);
}

/*
		File Name: CBRSim.h
		The CCBRsim class simulates the Cluster Based Routing
		protocol. It uses the CCBRNode class for the CBR nodes.
		The functions declared in this class are implemented in
		the CBRSim.cpp file.
*/
# pragma once
# include "CBRNode.h"
# include <math.h>
# include <vector>
# include <iostream>
# include <fstream>
using namespace std;
/*
		The CCBRSim class has variables and functions that support the
		simulation of CBR protocol. It has variables to collect the
		simulation results and parameters to specify the simulation
		constraints.
*/
class CCBRSim
{
public:
		vector <CCBRNode> Nodelist; //Node list
		vector <int> listGlobal2; //Temp list for nodes.
		long lClock; //Simulator clock.
		int iNonodesdied; //Number of nodes dead.
		int iRadius; //Radius of nodes communication.
		int iMaxnodes; //Maximum number of nodes in network.
		int iMaxdesiredspeed; //Nodes desired speed.
		int iBoundaryX; //Boundary X-coordinate.
		int iBoundaryY; //Boundary Y-coordinate.
		int iMaxsecurity;//Maximum security for a node.
		long iMaxpowerlevel;//Intial power level of a node.
		//Simulation results collection.
		long lTotal_message_succesful; //Messages successfull.
		unsigned long lTotal_overhead; //Total overhead for simulation.
		long lTotal_message_dropped; //Messages dropped in simulation.
		long lTotal_message_sent; //Messages sent during simulation.
		long lTotal_power; //Total power of all the nodes.
		long lTotal_transattempt; //The transmission attempts.
		long lTotal_message_lost; // Messages lost in the network.
		//Functions to support simulation.
		CCBRSim(void);
		void Send_beacon_to_nodes(int iId);
		void Sendmessage(CMessage_cbr Message);
		void Sendrreq(CMessage_cbr Message);
		void CBRsimulation(ostream &out, unsigned int lSeed, double dPercent);
		void CBRsimtime(ostream &out, unsigned int lSeed, long lMaxc);
};
```

```
/*
        File Name: CBRSim.cpp
        This file has the implementation of functions belonging to
        the CCBRSim class present in the CBRSim.h file.
*/
# include "CBRSim.h"
// Default constructor of the CCBRSim class.
CCBRSim::CCBRSim(void)
{
        lClock=0;
        iNonodesdied=0;
        iRadius=0;
        iMaxnodes=0;
        iMaxdesiredspeed=0;
        iBoundaryX=0;
        iBoundaryY=0;
        iMaxsecurity=0;
        iMaxpowerlevel=0;
        lTotal_message_succesful=0;
        lTotal_overhead=0;
        lTotal_message_dropped=0;
        lTotal_message_sent=0;
        lTotal_power=0;
        lTotal_transattempt=0;
        lTotal_message_lost =0;
}
/*
        The function is used to send beacons between nodes. It has
        a node ID as input. It sends beacons to all the nodes that
        are physically surrounding the input node.
*/
void CCBRSim::Send_beacon_to_nodes(int iId)
{
        unsigned int iTemp;
        // Identify the node in list.
        for(iTemp=0; iTemp < Nodelist.size(); iTemp++)
        {
                if(Nodelist[iTemp].iNodeid== iId)
                {
                        break;
                }
        }
        //Create a beacon message.
        CMessage_cbr Newmessage;
        Newmessage.iSourceid = iId;
        Newmessage.iMessagetype = 0;
        Newmessage.iMessageid = Nodelist[iTemp].iCurstatus;
        Newmessage.iSenderid = Nodelist[iTemp].iClusterid;
        Newmessage.iTimestamp  = lClock;
        Nodelist[iTemp].iLastbeacon=lClock;
        int xother, yother, xthis, ythis;
  double Caldistance;
  //Get the location of the node.
  xthis = Nodelist[iTemp].iCurrloc_X;
  ythis = Nodelist[iTemp].iCurrloc_Y;
  unsigned int iTemp1;
  for(iTemp1=0; iTemp1 <Nodelist.size(); iTemp1++)
  {
          if(iTemp != iTemp1)
          {
                  //Check it is not the beacon sending node.
                  //Calculate the distance.
                  xother = Nodelist[iTemp1].iCurrloc_X;
                  yother = Nodelist[iTemp1].iCurrloc_Y;
                  Caldistance=sqrt(double((xother-xthis)*(xother-xthis) + (yother-ythis)*(yother-ythis)));
                  if(iRadius >= int(Caldistance))
                  {
                          //Send beacon.
                          Newmessage.iDestinationid = iTemp1;
                          Newmessage.iReceiverid  = iTemp1;
                          Nodelist[iTemp1].listMessage.push_back(Newmessage);
                  }
```

```
            }
    }
}
/*
        The function is used to send the input message between
        two nodes. The sender and receiver nodes are identified
        from the message. if the nodes are out of reach then
        the message is ignored.
*/
void CCBRSim::Sendmessage(CMessage_cbr Message)
{
  int xother, yother, xthis, ythis;
  double Caldistance;
  int iTonodeid, iSendnodeid;
  iTonodeid = Message.iReceiverid; //Receiver node
  iSendnodeid = Message.iSenderid; //Sender node.
  unsigned int iTemp;
  //Locate sender in the list.
  for(iTemp = 0; iTemp < Nodelist.size(); iTemp ++)
  {
        if(iSendnodeid == Nodelist[iTemp].iNodeid)
        {
                break;
        }
  }
  //Get sender current location.
   xthis = Nodelist[iTemp].iCurrloc_X;
   ythis = Nodelist[iTemp].iCurrloc_Y;
   //Locate receiver node in the list.
  for(iTemp = 0; iTemp < Nodelist.size(); iTemp ++)
  {
        if(iTonodeid == Nodelist[iTemp].iNodeid)
        {
                break;
        }
  }
  //Calculate distance between sender and receiver.
  xother = Nodelist[iTemp].iCurrloc_X;
  yother = Nodelist[iTemp].iCurrloc_Y;
  Caldistance=sqrt(double((xother-xthis)*(xother-xthis) + (yother-ythis)*(yother-ythis)));
  if(int(Caldistance) < iRadius)
  {
        //if they are within range then send message.
        Nodelist[iTemp].listMessage.push_back(Message);
  }
  else
  {
        //Ignore message as out of range.
        lTotal_message_lost++;
  }
}
/*
        The function sends RREQ message to all the node present
        in the temporary list. The input message is forwarded to a
        node using the Sendmessage function.
*/
void CCBRSim::Sendrreq(CMessage_cbr Message)
{
  unsigned int iTemp;
  for(iTemp=0; iTemp < listGlobal2.size(); iTemp++)
  {
        Message.iReceiverid  = listGlobal2[iTemp];
        Sendmessage(Message);
  }
  listGlobal2.clear();
}
/*
        The CBRsimulation function simulates CBR protocol until the
        input percentage of nodes are dead. The function has three inputs
        output file stream, random function seed value and percentage of
        node in network to be dead.
*/
```

```
void CCBRSim::CBRsimulation(ostream &out, unsigned int lSeed, double dPercent)
{
	//Initially assign the desired number of nodes with the given parameters.
	unsigned int iNodes;
	srand(lSeed);
	for(iNodes=0; int(iNodes) < iMaxnodes; iNodes++)
	{
		CCBRNode Tempnode;
		Tempnode.iNodeid= iNodes;
		Tempnode.iMaxspeed = iMaxdesiredspeed;
		Tempnode.iBoundary_X = iBoundaryX;
		Tempnode.iBoundary_Y = iBoundaryY;
		Tempnode.iCurrloc_X = rand()%iBoundaryX;
		Tempnode.iCurrloc_Y = rand()%iBoundaryY;
		Tempnode.iPowerlevel = iMaxpowerlevel;
		Tempnode.iSecuritylevel = rand()%iMaxsecurity;
		Nodelist.push_back(Tempnode);
	}
	lClock=0;
	while (iNonodesdied < (iMaxnodes * (dPercent/100)))
	{
		//When the number of nodes dead constraint is satisfied.
		lClock++;
		//For each node in the network.
		for(iNodes=0; iNodes < Nodelist.size(); iNodes++)
		{
			if((Nodelist[iNodes].iPowerlevel <=0)&&(Nodelist[iNodes].bCounted == false))
			{
				//if the power loss in the node was detected first time.
				Nodelist[iNodes].bCounted = true;
				iNonodesdied++;
			}
			if(Nodelist[iNodes].iPowerlevel >0)
			{
				//if node has power in it.
			if(lClock > 5)
			{
				//To give time for the network to settle down.
				Nodelist[iNodes].Set_future();
				Nodelist[iNodes].Move_node();
				Nodelist[iNodes].Determine_Send_Message(lClock, int(Nodelist.size()));
				CMessage_cbr Tempmessage = Nodelist[iNodes].Send_Message(lClock);
				if(Tempmessage.iMessagetype != 0)
				{
					if(Tempmessage.iMessagetype ==1)
					{
						//Foward the RREQ message.
						Nodelist[iNodes].Forwardrreq(Tempmessage);
						unsigned int iVec;
						for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
						{
							listGlobal2.push_back(Nodelist[iNodes].listGlobal[iVec]);
						}
						Sendrreq(Tempmessage);
					}
					else
					{
						//Send other messages.
						Sendmessage(Tempmessage);
					}
				}
			}
			bool bBeacon;
			//Send a beacon.
			bBeacon=Nodelist[iNodes].Send_beacon(lClock);
			if(bBeacon== true)
			{
				//if node decides to send beacon.
				Send_beacon_to_nodes(iNodes);
			}
			CMessage_cbr Tempmessage;
			while(1)
```

```
                    {
                            //Read the messages in the list.
                            Tempmessage = Nodelist[iNodes].Read_Message(lClock);
                            if(Tempmessage.iMessagetype == 6)
                            {
                                    // No more messages to read.
                                    break;
                            }
                            if(Tempmessage.iMessagetype == 1)
                            {
                                    // Need to send RREQ packet.
                                    Nodelist[iNodes].Forwardrreq(Tempmessage);
                                    unsigned int iVec;
                                    for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
                                    {
                                            listGlobal2.push_back(Nodelist[iNodes].listGlobal[iVec]);
                                    }
                                    Sendrreq(Tempmessage);
                            }
                            if(Tempmessage.iMessagetype != 5)
                            {
                                    //Forward other packets, ignore the message if type =5.
                                    Sendmessage(Tempmessage);
                            }
                    }
                    //Update the tables and check the cluster consistency.
                    Nodelist[iNodes].Updatealltables(lClock);
                    Nodelist[iNodes].Decide_cluster();

            }
            else
            {
                    Nodelist[iNodes].listMessage.clear();
            }
            }
    }
    //Collect simulation results.
  for(iNodes=0; iNodes < Nodelist.size(); iNodes++)
  {
        lTotal_message_succesful= lTotal_message_succesful + Nodelist[iNodes].lMessage_succes;
        lTotal_overhead= lTotal_overhead + Nodelist[iNodes].lOverhead;
        lTotal_message_dropped= lTotal_message_dropped + Nodelist[iNodes].lMessage_drop;
        lTotal_message_sent= lTotal_message_sent + Nodelist[iNodes].lMessage_sent;
        lTotal_power= lTotal_power + Nodelist[iNodes].iPowerlevel;
        lTotal_transattempt = lTotal_transattempt + Nodelist[iNodes].lTotaltrans;
  }
  //Output the simulation results.
  out<<" Cluster Based Routing Protocol Simulation";
  out<<"\n";
  out<<"Number of Nodes: "<<iMaxnodes<<"\n";
  out<<"Simulation Time: "<<lClock<<"\n";
  out<<"Number of Nodes Dead: "<<iNonodesdied<<"\n";
  out<<"Average Power Left of a Node: "<<double(lTotal_power / iMaxnodes)<<"\n";
  out<<"Number of Message Sent: "<<lTotal_message_sent<<"\n";
  out<<"Number of Message reached Destination: "<<lTotal_message_succesful<<"\n";
  out<<"Number of Message dropped: "<<lTotal_message_dropped<<"\n";
  out<<"Total Overhead: "<<lTotal_overhead<<"\n";
  out<<"Total Transmission attempts: "<<lTotal_transattempt<<"\n";
  out<<"Total Messages lost in Network: "<<lTotal_message_lost<<"\n";
  out<<"% of Messages transmited succesufuly: "<<((double(lTotal_message_succesful)/double(lTotal_message_sent))*100)<<"\n";
  out<<"\n";
}
/*
        The CBRsimtime funciton is simular to the above function with the
        difference in the constraint. The maximum clock value is the constraint
        overhere. The simulation is performed until that clock value is reached.
*/
void CCBRSim::CBRsimtime(ostream &out, unsigned int lSeed, long lMaxc)
{
// Initially assign the desired number of nodes with the given parameters.
        unsigned int iNodes;
        srand(lSeed);
```

```
for(iNodes=0; int(iNodes) < iMaxnodes; iNodes++)
{
	CCBRNode Tempnode;
	Tempnode.iNodeid= iNodes;
	Tempnode.iMaxspeed = iMaxdesiredspeed;
	Tempnode.iBoundary_X = iBoundaryX;
	Tempnode.iBoundary_Y = iBoundaryY;
	Tempnode.iCurrloc_X = rand()%iBoundaryX;
	Tempnode.iCurrloc_Y = rand()%iBoundaryY;
	Tempnode.iPowerlevel = iMaxpowerlevel;
	Tempnode.iSecuritylevel = rand()%iMaxsecurity; //randomly assign security levels.
	Nodelist.push_back(Tempnode);
}
lClock=0;
while (lClock < lMaxc)
{
	//Untill the clock constriant is reached.
	lClock++;
	//for each node present in network.
	for(iNodes=0; iNodes < Nodelist.size(); iNodes++)
	{
		if((Nodelist[iNodes].iPowerlevel <=0)&&(Nodelist[iNodes].bCounted == false))
		{
			//if node was not previously counted.
			Nodelist[iNodes].bCounted = true;
			iNonodesdied++;
		}
		if(Nodelist[iNodes].iPowerlevel >0)
		{
			//if node has power left.
		if(lClock > 5)
		{
			//To give time for the network to settle down.
			Nodelist[iNodes].Set_future();
			Nodelist[iNodes].Move_node();
			Nodelist[iNodes].Determine_Send_Message(lClock, int(Nodelist.size()));
			CMessage_cbr Tempmessage = Nodelist[iNodes].Send_Message(lClock);
			if(Tempmessage.iMessagetype != 0)
			{
				if(Tempmessage.iMessagetype ==1)
				{
					//Send RREQ packet.
					Nodelist[iNodes].Forwardrreq(Tempmessage);
					unsigned int iVec;
					for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
					{
						listGlobal2.push_back(Nodelist[iNodes].listGlobal[iVec]);
					}
					Sendrreq(Tempmessage);
				}
				else
				{
					Sendmessage(Tempmessage);
				}
			}
		}
		bool bBeacon;
		//Decide if to send beacon.
		bBeacon=Nodelist[iNodes].Send_beacon(lClock);
		if(bBeacon== true)
		{
			Send_beacon_to_nodes(iNodes);
		}
		CMessage_cbr Tempmessage;
		while(1)
		{
			//Read messages from its list.
			Tempmessage = Nodelist[iNodes].Read_Message(lClock);
			if(Tempmessage.iMessagetype == 6)
			{
				//No more messages to read.
				break;
```

```
					}
					if(Tempmessage.iMessagetype == 1)
					{
						//Send RREQ packet.
						Nodelist[iNodes].Forwardrreq(Tempmessage);
						unsigned int iVec;
						for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
						{
							listGlobal2.push_back(Nodelist[iNodes].listGlobal[iVec]);
						}
						Sendrreq(Tempmessage);
					}
					if(Tempmessage.iMessagetype != 5)
					{
						//ignore if message type =5 or send it.
						Sendmessage(Tempmessage);
					}
				}
				//Update tables and check the consistency of clustering.
				Nodelist[iNodes].Updatealltables(lClock);
				Nodelist[iNodes].Decide_cluster();

			}
			else
			{
				Nodelist[iNodes].listMessage.clear();
			}
			}
		}
		//simulation complete, collect results.
  for(iNodes=0; iNodes < Nodelist.size(); iNodes++)
  {
		lTotal_message_succesful= lTotal_message_succesful + Nodelist[iNodes].lMessage_succes;
		lTotal_overhead= lTotal_overhead + Nodelist[iNodes].lOverhead;
		lTotal_message_dropped= lTotal_message_dropped + Nodelist[iNodes].lMessage_drop;
		lTotal_message_sent= lTotal_message_sent + Nodelist[iNodes].lMessage_sent;
		lTotal_power= lTotal_power + Nodelist[iNodes].iPowerlevel;
		lTotal_transattempt = lTotal_transattempt + Nodelist[iNodes].lTotaltrans;
  }
  //Output results.
  out<<" Cluster Based Routing Protocol Simulation";
  out<<"\n";
  out<<"Number of Nodes: "<<iMaxnodes<<"\n";
  out<<"Simulation Time: "<<lClock<<"\n";
  out<<"Number of Nodes Dead: "<<iNonodesdied<<"\n";
  out<<"Average Power Left of a Node: "<<double(lTotal_power / iMaxnodes)<<"\n";
  out<<"Number of Message Sent: "<<lTotal_message_sent<<"\n";
  out<<"Number of Message reached Destination: "<<lTotal_message_succesful<<"\n";
  out<<"Number of Message dropped: "<<lTotal_message_dropped<<"\n";
  out<<"Total Overhead: "<<lTotal_overhead<<"\n";
  out<<"Total Transmission attempts: "<<lTotal_transattempt<<"\n";
  out<<"Total Message lost in Network: "<<lTotal_message_lost<<"\n";
  out<<"% of Messages transmited succesufuly: "<<((double(lTotal_message_succesful)/double(lTotal_message_sent))*100)<<"\n";
  out<<"\n";
}

/*
		File Name: SCBRNode.h
		The file has declaration of classes required for supporting a
		Sector Based Clustering and Routing (SBCR) protocol node. The
		functions present in these classes are implemented in
		SBCRNode.cpp file.
*/
# pragma once
# include "Node.h"
 /*
		The CAdajcency_sbcr structure is used for maintaining the neighbor
		information for each node. When a node receives a beacon from
		another node then an entry is added or updated in the adjacency
		table. Regular updates are made to have current neighbor information.
 */
class CAdjacency_sbcr
```

```
{
public:
        int iNeighborid; //Neighbor node ID.
        int iRole; // Role of the neighbor node.
        long iTimestamp; //Time stamp
        int iAdjvalue; //Adajaceny value with respect to neighbor node.
        int iSector; //The sector it is present in.
        int iSecuritylevel; //The security level it has.
        //Functions
        CAdjacency_sbcr(void);
        CAdjacency_sbcr(const CAdjacency_sbcr &x1);
        CAdjacency_sbcr & operator= (CAdjacency_sbcr x1);
        void Setadjacency(int iNeigh, int iR, long iTime, int iAdjv, int iSec, int iSecurlevel);
};
/*
        The CMember class is used by the cluster heads in the SBCR protocol
        to maintain the members present in their cluster.
*/
class CMember
{
public:
        int iMemberid; //Member node id.
        int iRole; //Member role.
        int iSector; //Sector it belongs to.
        int iSecuritylevel; //Security level it has.
        //Functions.
        CMember(void);
        CMember(const CMember &x1);
        CMember & operator= (CMember x1);
        void Setmember(int iMem, int iR, int iSec, int iSecurlevel);
};
/*
        The CRreqlistsbcr class is used to maintian the list of RREQ
        messages boradcasted by other nodes. When a node receives
        RREQ message it check if an instance of this is already
        present. if not then it adds an entry in the table and
        forwards the RREQ message to other nodes.
*/
class CRreqlistsbcr
{
public:
        int iSourceid; //Source id.
        int iRreqid; //RREQ message id.
        long iTimestamp; //Timestamp.
        //Functions.
        CRreqlistsbcr(void);
        CRreqlistsbcr(const CRreqlistsbcr &x1);
        CRreqlistsbcr & operator=(CRreqlistsbcr x1);
        void Setrreq(int iSour, int iRreq, long iTime);
};
/*
        The CRoutes class is used for maintaining the list of routes
        information obtained for a single transmission by a node.
*/
class CRoutes
{
public:
        vector <int> listRoute; //single route information
        int iRouteid; //route id.
        //functions.
        CRoutes(void);
        CRoutes(const CRoutes &x1);
        CRoutes & operator= (CRoutes x1);
};
/*
        The CRerror class is used to maintain all the error messages
        issued by a node. This is done to avoid sending a error message
        for a perfect route.
*/
class CRerror
{
public:
```

```
		int iRouteid; //Route id.
		int iSourceid; //Source node id.
		int iDestinationid; //Destination id.
		long iTimestamp; //Timestamp.
		CRerror(void);
		CRerror(const CRerror &x1);
		CRerror & operator= (CRerror x1);
};
/*
		The CMessage_sbcr class has multi-purpose use in this simulation.
   It is used as a message data structure- a medium to send messages
		between the nodes in the network. Each node has a list to store
		all the messages it receives and later reads them. The structure
		is also used as return type variable to exchange parameter values
		between the functions.
*/
class CMessage_sbcr
{
public:
		int iMessageid; //Message id.
		int iMessagetype; //Message type.
		int iDestinationid; //Destination id.
		int iSourceid; //Source id.
		int iSenderid; //Sender id.
		int iReceiverid; //Receiver id.
		long iTimestamp; //Timestamp.
		int iRouteid; //Route id.
		int iLocx; //Source location X-coordinate.
		int iLocy; //Source location Y-coordinate.
		vector <int> listRoute; //Route information.
		int iLocroute; //Current location in route.
		int iNoofhops; //Number of hops.
		//Functions.
		CMessage_sbcr(void);
		CMessage_sbcr(const CMessage_sbcr &x1);
		CMessage_sbcr & operator= (CMessage_sbcr x1);
		int Returnrouteloc(void);
		void Addtoroute(int nodeid);
		void Createrreq(int iRid, int iDestinid, int iSourid, long iTime, int iLx, int iLy);
};
/*
		The CSBCRNode class represents the Sector Based Clustering
		and Routing protocol node in the network. It inherits the
		CNode class and includes other variables and functions to
		support SBCR node.
*/
class CSBCRNode : public CNode
{
public:
		int iSectors; //Number of sectors present in the network.
		int iNodespersector; //Nodes per sector as members.
		int iCurdestin; //Current destination.
		int iCurpacket; //Current packet transmitted.
		int iCurstatus; //Status of node.
		bool bTransmitdata; //if transmitting data.
		bool bRoutereq; //if route requested.
		bool bRouteest; //if route established.
		bool bTransferdata; //if involved in any data transmission.
		long iLasttransfertime; //last time data forwarded or sent.
		int iRreqid; //RREQ packet id.
		int iReqcount; //RREQ number of times requested.
		long iInitialtime; //RREQ requested time
		int iNumberofpackets; //Number of packets to be transmitted.
		int iNumberofinvitations; //Number of invitations received.
		int iLastroute; //Last route that was used.
		int iNumberpaths; //Number of paths found.
		int iNotransmisions; //Number of transmission.
		bool bDirectroute; //if transmission is single hop.
		list <CRoutes> listRcollection; //Route list.
		list <CMessage_sbcr> listMessage; //Message list.
		list <CAdjacency_sbcr> listAdjacency; //Adjacency list.
		list <CMember> listMember; //Member list.
```

```
		list <CRreqlistsbcr> listRreq; //RREQ list.
		list <CRerror> listRerr; //Rerr list.
		vector <int> listGlobal; //Temprorary list.
		//Functions.
		bool Determine_Send_Message(long iTime, int iMaxnodes);
		CMessage_sbcr Send_Message(long iTime);
		CMessage_sbcr Read_Message(long iTime);
		int Sectorassociation(int iLoc1x, int iLoc1y, int iLoc2x, int iLoc2y);
		void Add_adjacency(int iNeigh, int iSector, int iR, long iTime, int iSeclevel);
		void Update_adjacency(long iTime);
		bool Add_Rreqlist(int iSour, int iRid, long iTime);
		void Update_Rreqlist(long iTime);
		void Add_member(int iMid, int iS, int iR, int iSeclevel);
		CRoutes Return_route(int iRid);
		int Decide_nextnode(int iL1x, int iL1y, int iL2x, int iL2y, int iDistance, int iDiscovered, int iDestinid, int iSenderid);
		void Delete_route(int iId, int iSender);
		bool Is_member(int iId);
		int Automataclusterdecision(long iTime, int iMaxsecuritylevel);
		bool Sector_hasnode(int iSecid);
		int Weight(int iSec1, int iSec2);
		void Update_member(void);
		bool Is_adjacent(int iId);
		CSBCRNode(void);
		CSBCRNode(const CSBCRNode &x1);
		CSBCRNode & operator = (CSBCRNode x1);
		bool Add_Rerrorlist(int Rid, int iSour, int iDestin, long iTime);
		void Update_rerror(long iTime);
		void Forwardrreq(CMessage_sbcr Message);
		CMessage_sbcr Invitemember(void);
		void Forwardinitialrrep(int iSectorto);
		void Updatealltables(long iTime);

};

/*
		File Name: SBCRNode.cpp
		The file has the implementation of functions belonging to
		classes present in SBCRNode.h file.
*/
# include "SBCRNode.h"

//Default constructor of CAdjacency_sbcr class.
CAdjacency_sbcr::CAdjacency_sbcr(void)
{
		iNeighborid =0;
		iRole =0;
		iTimestamp =0;
		iAdjvalue =0;
		iSector =0;
		iSecuritylevel =0;
}
//Copy constructor of CAdjacency_sbcr class.
CAdjacency_sbcr::CAdjacency_sbcr(const CAdjacency_sbcr &x1)
{
		iNeighborid = x1.iNeighborid;
		iRole = x1.iRole;
		iTimestamp = x1.iTimestamp;
		iAdjvalue = x1.iAdjvalue;
		iSector = x1.iSector;
		iSecuritylevel = x1.iSecuritylevel;
}
//= Operator overloading for CAdjacency_sbcr class.
CAdjacency_sbcr & CAdjacency_sbcr::operator =(CAdjacency_sbcr x1)
{
		iNeighborid = x1.iNeighborid;
		iRole = x1.iRole;
		iTimestamp = x1.iTimestamp;
		iAdjvalue = x1.iAdjvalue;
		iSector = x1.iSector;
		iSecuritylevel = x1.iSecuritylevel;
		return(*this);
}
```

```
/*
        Function intializes the CAdjacency_sbcr class object to the given
        parameters. it has the Neighbor id, Role, Timestamp, Adjacency value,
        sector it belongs to, and security level as input.
*/
void CAdjacency_sbcr::Setadjacency(int iNeigh, int iR, long iTime, int iAdjv, int iSec, int iSecurlevel)
{
        iNeighborid = iNeigh;
        iRole = iR;
        iTimestamp = iTime;
        iAdjvalue = iAdjv;
        iSector = iSec;
        iSecuritylevel = iSecurlevel;
}
//Default constructor of CMember class.
CMember::CMember(void)
{
        iMemberid=0;
        iRole =0;
        iSector=0;
        iSecuritylevel=0;
}
//Copy constructor of CMember class.
CMember::CMember(const CMember &x1)
{
        iMemberid = x1.iMemberid;
        iRole = x1.iRole;
        iSector = x1.iSector;
        iSecuritylevel = x1.iSecuritylevel;
}
//= Operator Overloading for CMember class.
CMember & CMember::operator=(CMember x1)
{
        iMemberid = x1.iMemberid;
        iRole = x1.iRole;
        iSector = x1.iSector;
        iSecuritylevel = x1.iSecuritylevel;
        return(*this);
}
/*
         The function intializes the CMember class object to the given
         input values. It has inputs Member id, role , Sector, and
         security level.
*/
void CMember::Setmember(int iMem,int iR, int iSec, int iSecurlevel)
{
        iMemberid = iMem;
        iRole = iR;
        iSector = iSec;
        iSecuritylevel = iSecurlevel;
}
//Default constructor of CRreqlistsbcr class.
CRreqlistsbcr::CRreqlistsbcr(void)
{
        iSourceid =0;
        iRreqid =0;
        iTimestamp=0;
}
//Copy constructor of CRreqlistsbcr class.
CRreqlistsbcr::CRreqlistsbcr(const CRreqlistsbcr &x1)
{
        iSourceid = x1.iSourceid;
        iRreqid = x1.iRreqid;
        iTimestamp = x1.iTimestamp;
}
//= Operator Overloading for the CRreqlistsbcr class.
CRreqlistsbcr& CRreqlistsbcr::operator =(CRreqlistsbcr x1)
{
        iSourceid = x1.iSourceid;
        iRreqid = x1.iRreqid;
        iTimestamp = x1.iTimestamp;
        return(*this);
```

```
}
//Default constructor of CRoutes class.
CRoutes::CRoutes(void)
{
		iRouteid=0;
}
//Copy Constructor of Croutes class.
CRoutes::CRoutes(const CRoutes &x1)
{
		listRoute = x1.listRoute;
		iRouteid = x1.iRouteid;
}
//= Operator Overloading of Croutes class.
CRoutes& CRoutes::operator=(CRoutes x1)
{
		listRoute = x1.listRoute;
		iRouteid = x1.iRouteid;
		return(*this);
}
//Default constructor of the CMessage_sbcr class.
CMessage_sbcr::CMessage_sbcr(void)
{
		iMessageid =0;
		iMessagetype =0;
		iSenderid =0;
		iSourceid =0;
		iReceiverid =0;
		iDestinationid =0;
		iLocroute =0;
		iTimestamp =0;
		iLocx=0;
		iLocy=0;
		iRouteid=0;
		iNoofhops=0;
}
//Copy constructor of the CMessage_sbcr class.
CMessage_sbcr::CMessage_sbcr(const CMessage_sbcr &x1)
{
		iMessageid = x1.iMessageid;
		iMessagetype =x1.iMessagetype;
		iSenderid = x1.iSenderid;
		iSourceid = x1.iSourceid;
		iReceiverid = x1.iReceiverid;
		iDestinationid = x1.iDestinationid;
		iLocroute = x1.iLocroute;
		listRoute = x1.listRoute;
		iTimestamp = x1.iTimestamp;
		iLocx = x1.iLocx;
		iLocy = x1.iLocy;
		iRouteid = x1.iRouteid;
		iNoofhops = x1.iNoofhops;

}
//= Operator Overloading for the CMessage_sbcr class.
CMessage_sbcr & CMessage_sbcr::operator=(CMessage_sbcr x1)
{
		iMessageid = x1.iMessageid;
		iMessagetype =x1.iMessagetype;
		iSenderid = x1.iSenderid;
		iSourceid = x1.iSourceid;
		iReceiverid = x1.iReceiverid;
		iDestinationid = x1.iDestinationid;
		iLocroute = x1.iLocroute;
		listRoute = x1.listRoute;
		iTimestamp = x1.iTimestamp;
		iLocx = x1.iLocx;
		iLocy = x1.iLocy;
		iRouteid = x1.iRouteid;
		iNoofhops = x1.iNoofhops;
		return(*this);
}
/*
```

```
		The function returns the next node present in the route. if the
		node reference is invalid the it returns -1.
*/
int CMessage_sbcr::Returnrouteloc(void)
{
		if(iLocroute <0)
		{
				return(-1);
		}
		if(iLocroute >= int(listRoute.size()))
		{
				return(-1);
		}
		return(listRoute[iLocroute]);
}
/*
		The function adds a node to the current route list.
*/
void CMessage_sbcr::Addtoroute(int nodeid)
{
		listRoute.push_back(nodeid);
}
/*
		The function creates a RREQ packet and initializes to the given
		input. The RREQ id, destination id, source id, time stamp, Location
		x-coordinate, and Location Y- coordinate are given as input.
*/
void CMessage_sbcr::Createrreq(int iRid, int iDestinid, int iSourid, long iTime, int iLx, int iLy)
{
		iMessagetype =1;
		iMessageid= iRid;
		iDestinationid = iDestinid;
		iSourceid = iSourid;
		iTimestamp = iTime;
		iLocx = iLx;
		iLocy = iLy;
}
/*
		The function Determine_Send_Message decides if a node
		wants to transmit data to another node. it has the current
		clock value and maximum number of nodes present as inputs.
		if a node randomly decides to transmit data then the
		function returns true else false. The function also decides
		the destination node and the number of packets to be
		transmitted.
*/
bool CSBCRNode::Determine_Send_Message(long iTime, int iMaxnodes)
{
		if(bTransmitdata==false)
		{
		int iProb, iTempnode, iMul;
		iProb = rand()%10;
   if(iProb < 8)
		{
				//When no data is to be sent
				return(false);
		}
		else
		{
				 //When data is to be sent.
     iTempnode = rand()%iMaxnodes; //Determine the destination id.
		 if(iTempnode == iNodeid)
		 {
				 if(iTempnode < (iMaxnodes-2))
				 {
						 iTempnode++;
				 }
				 else
				 {
						 if(iTempnode == 0)
						 {
								 iTempnode++;
```

```
					}
					else
					{
							iTempnode--;
					}
				}
		}
		//Reset the values.
		iCurdestin = iTempnode;
		iMul = rand()%10;
		iMul++;
		iNumberofpackets = 100* iMul;
		bRouteest = false;
		bRoutereq=false;
		bTransmitdata = true;
		iCurpacket =0;
		lTotaltrans++;
		listRcollection.clear();
		return(true);
   }
		}
		return(false);
}
/*
		The function returns the sector value the neighbor node belongs to. It
		has inputs iLoc1x and iLoc1y the location of the current node and iLoc2x
		and iLoc2y the location of neighbor node. The function calculates and
		returns in which sector of the current node the neighbor node is present.
*/
int CSBCRNode::Sectorassociation(int iLoc1x, int iLoc1y, int iLoc2x, int iLoc2y)
{
		int i;
		double dTheta;
		//Calculate the angle.
		if((iLoc2x- iLoc1x)==0)
		{
				if(iLoc2y < iLoc1y)
				{
						dTheta = 3.142/2.0;
				}
				if(iLoc2y > iLoc1y)
				{
						dTheta = 1.5 * 3.142;
				}
				if(iLoc2y== iLoc1y)
				{
						dTheta=0;
				}
		}
		else
		{
				dTheta=atan(double((iLoc2y-iLoc1y)/(iLoc2x-iLoc1x)));
				if(dTheta < 0)
				{
						dTheta = dTheta + 6.284;
				}
				if(dTheta > 6.284)
				{
						dTheta = dTheta - 6.284;
				}
		}
		double dAngle1, dAngle2, dSectorangle;
		dSectorangle= double(6.284 / iSectors);
		dAngle1=0.0;
		dAngle2= dSectorangle;
		//Decide the sector values.
		for(i=1; i<= iSectors; i++)
		{
   if((dAngle1 <= dTheta) && (dTheta <= dAngle2))
		{
				return(i);
		}
```

```
		dAngle1= dAngle1+ dSectorangle;
		dAngle2= dAngle2 + dSectorangle;
	}
	return(1);
}
/*
	The function creates a CAdjacency_sbcr object and adds it to the
	Adjacency list. This function is called when a beacon is received
	by the node. if the neighbor information is already present in the
	list then it is updated else a new entry is added.
*/
void CSBCRNode::Add_adjacency(int iNeigh, int iSector, int iR, long iTime, int iSeclevel)
{
	list <CAdjacency_sbcr>::iterator itPointer;
	int iPrevadjvalue;
	iPrevadjvalue=0;
	//Check if the node entry is already present.
	for(itPointer= listAdjacency.begin(); itPointer!= listAdjacency.end(); itPointer++)
	{
		CAdjacency_sbcr Tempadjacency =*itPointer;
		if(Tempadjacency.iNeighborid == iNeigh)
		{
			iPrevadjvalue= Tempadjacency.iAdjvalue;
			listAdjacency.erase(itPointer);
			break;
		}
	}
	//Add new entry in adjacency..
	CAdjacency_sbcr Tempadj;
	Tempadj.iAdjvalue = iPrevadjvalue+1;
	Tempadj.iNeighborid  = iNeigh;
	Tempadj.iRole = iR;
	Tempadj.iTimestamp = iTime;
	Tempadj.iSector = iSector;
	Tempadj.iSecuritylevel = iSeclevel;
	listAdjacency.push_back(Tempadj);
	//if the node is also a member then update information in member table.
	if(Is_member(iNeigh)==true)
	{
		Add_member(iNeigh, iSector, iR, iSeclevel);
	}
}
/*
	The function is used for storing current information in the
	adjacency list. Outdated information is regularly removed from
	the list. if an entry in the adjacency is timed out then it is
	removed. Different timeout values are used depending on nodes
	mobility.
*/
void CSBCRNode::Update_adjacency(long iTime)
{
	list <CAdjacency_sbcr>::iterator itPointer;
	list <CAdjacency_sbcr> listTemp;
	int iT;
	if(bMobility == true)
	{
		iT = 10;
	}
	else
	{
		iT = 15;
	}
	//Move to a temp list.
	while(!listAdjacency.empty())
	{
		itPointer= listAdjacency.begin();
		CAdjacency_sbcr Curr_node = *itPointer;
		if((iTime-Curr_node.iTimestamp)<=iT)
		{
			//Entry is not expired.
			listTemp.push_back(Curr_node);
		}
```

```
                listAdjacency.pop_front();
        }
        //Move from temp list to adjacency list.
        while(!listTemp.empty())
        {
                itPointer = listTemp.begin();
                CAdjacency_sbcr Curr_node = *itPointer;
          listAdjacency.push_back(Curr_node);
                listTemp.pop_front();
        }
}

/*
        The function adds a RREQ entry in the RREQ list. It has the
        Source id, RREQ id, and timestamp as input. if an entry was
        previously present then it ignores it and does'nt forward the
        RREQ message to other nodes.
*/
bool CSBCRNode::Add_Rreqlist(int iSour, int iRid, long iTime)
{
        list <CRreqlistsbcr>::iterator itPointer;
        bool bFlag;
        bFlag=true;
        //Check if an entry exist
        for(itPointer = listRreq.begin(); itPointer != listRreq.end(); itPointer++)
        {
                CRreqlistsbcr Tempnode=*itPointer;
                if((Tempnode.iRreqid == iRid)&&(Tempnode.iSourceid == iSour))
                {
                        bFlag=false;
                        break;
                }
        }
        //Add an entry.
        if(bFlag==true)
        {
                CRreqlistsbcr Addnode;
                Addnode.iRreqid = iRid;
                Addnode.iSourceid = iSour;
                Addnode.iTimestamp = iTime;
                listRreq.push_back(Addnode);
        }
        return(bFlag);
}
/*
        The function is used to maintain updated values in the table.
        Any entry in the table with expired timestamp is removed.
*/
void CSBCRNode::Update_Rreqlist(long iTime)
{
        list <CRreqlistsbcr>::iterator itPointer;
        list <CRreqlistsbcr> listTemp;
        //Move to temp list.
        while(!listRreq.empty())
        {
                itPointer= listRreq.begin();
                CRreqlistsbcr Curr_node = *itPointer;
                if((iTime-Curr_node.iTimestamp)<=20)
                {
                         //Expired entry.
                        listTemp.push_back(Curr_node);
                }
                listRreq.pop_front();
        }
        listRreq.clear();
        //Move to Rreq list.
        while(!listTemp.empty())
        {
                itPointer = listTemp.begin();
                CRreqlistsbcr Curr_node = *itPointer;
          listRreq.push_back(Curr_node);
                listTemp.pop_front();
```

```
		}
}
/*
		The function is called when a cluster head adds a new member
		to its cluster. if an entry of this instance is already present
		then it is updated in the table.
*/
void CSBCRNode::Add_member(int iMid,int iS, int iR, int iSeclevel)
{
		CMember Tempmember;
		Tempmember.Setmember(iMid, iR, iS, iSeclevel);
		list <CMember>::iterator itPointer;
		//Check if an entry already exists.
		for(itPointer = listMember.begin(); itPointer != listMember.end(); itPointer++)
		{
				CMember Tempnode=*itPointer;
				if(Tempnode.iMemberid == Tempmember.iMemberid)
				{
						//remove the entry.
						listMember.erase(itPointer);
						break;
				}
		}
		//Add the entry.
				listMember.push_back(Tempmember);
}


/*
		The function returns a route from collection of routes. The
		input route id decides which route should be returned. if
		the route does'nt exist then the route id is set to -1.
*/
CRoutes CSBCRNode::Return_route(int iRid)
{
		list <CRoutes>::iterator itRoute;
		//Locate the route.
		for(itRoute = listRcollection.begin(); itRoute!= listRcollection.end(); itRoute++)
		{
				CRoutes Troute = *itRoute;
				if(Troute.iRouteid == iRid)
				{
						//Send route.
						return(Troute);
				}
		}
		//route does'nt exist.
		CRoutes Troute;
		Troute.iRouteid = -1;
		return(Troute);
}
//Default constructor of CRerror class.
CRerror::CRerror(void)
{
		iRouteid=0;
		iSourceid =0;
		iDestinationid =0;
		iTimestamp=0;
}
//Copy constructor of CRerror class.
CRerror::CRerror(const CRerror &x1)
{
		iRouteid = x1.iRouteid;
		iSourceid = x1.iSourceid;
		iDestinationid = x1.iDestinationid;
		iTimestamp = x1.iTimestamp;
}
//= Operator Overloading for CRerror class.
CRerror & CRerror::operator =(CRerror x1)
{
		iRouteid = x1.iRouteid;
		iSourceid = x1.iSourceid;
```

```
		iDestinationid = x1.iDestinationid;
		iTimestamp = x1.iTimestamp;
		return(*this);
}
/*
		The function Send_Message decides the packet to be sent. it has
		the current simulator clock value as the input. It returns a
		CAodvmessage that is read by network layer functions. The function
		decides whether to broadcast a RREQ packet. Once the route is
		established it transmits the packets one after the other. The
		function send a maximum of 3 RREQ packets for each transmission.
		if all three RREQ packets are timed out then transmission is
		aborted.
*/
CMessage_sbcr CSBCRNode::Send_Message(long iTime)
{
CMessage_sbcr Returnmessage;
if(bTransmitdata == true)
{
		//if node is transmitting data.
		Returnmessage.iDestinationid = iCurdestin;
		Returnmessage.iSourceid = iNodeid;

		if(bRouteest==false)
		{
			//if the Route was not established.
			if(bRoutereq==false)
			{
				//if route was not requested before.
				list <CAdjacency_sbcr>::iterator itPointer;
				//Check ajdacency list.
				for(itPointer = listAdjacency.begin(); itPointer != listAdjacency.end(); itPointer++)
				{
					CAdjacency_sbcr Currnode=*itPointer;
					if(Currnode.iNeighborid == iCurdestin)
					{
						//The destination is one of the adjacent neighbors.
						bRouteest=true;
						bTransferdata =true;
						bDirectroute = true;
						iLasttransfertime = iTime;
						CRoutes Temproute;
						Temproute.iRouteid =0;
						Temproute.listRoute.push_back(iNodeid);
						Temproute.listRoute.push_back(Currnode.iNeighborid);
						listRcollection.push_back(Temproute);
						Returnmessage.iMessageid = iCurpacket;
						Returnmessage.iMessagetype = 4; //For data message.
						Returnmessage.iLocroute =1;
						Returnmessage.Addtoroute(iNodeid);
						Returnmessage.Addtoroute(Currnode.iNeighborid);
						Returnmessage.iSenderid = iNodeid;
						Returnmessage.iReceiverid = Currnode.iNeighborid;
						Returnmessage.iTimestamp = iTime;
						Returnmessage.iRouteid = 0;
						iLastroute =0;
						iPowerlevel = iPowerlevel - 250;
						lMessage_sent++;
						return(Returnmessage);
					}
				}

				//if Destination is not present in the adjacency table.
				//A Route request is being sent.
				Returnmessage.Createrreq(iRreqid, iCurdestin,iNodeid, iTime,iCurrloc_X, iCurrloc_Y);
				Returnmessage.iNoofhops=0;
				Returnmessage.iSenderid= iNodeid;
				iPowerlevel = iPowerlevel - 75;
				bDirectroute = false;
				lOverhead++;
				iRreqid++;//Increment the Route reqeust id.
				iReqcount=0;
```

```
                        bRoutereq=true;
                        iInitialtime = iTime;
                        return(Returnmessage);
                }
                else
                {
                        if((iTime- iInitialtime)>10)
                        {
                                if(iReqcount <3)
                                {
                                //Send request again.
                                Returnmessage.Createrreq(iRreqid, iCurdestin,iNodeid, iTime,iCurrloc_X, iCurrloc_Y);
                                Returnmessage.iNoofhops=0;
                                Returnmessage.iSenderid= iNodeid;
                                Returnmessage.iReceiverid = iNodeid;
                                iPowerlevel = iPowerlevel - 75;
                                lOverhead++;
                                iRreqid++;
iReqcount++;
                                iInitialtime = iTime;
                                bDirectroute = false;
                                return(Returnmessage);
                                }
                                else
                                {
                                        //Cancel the data transmission request.
                                        bTransmitdata =false;
                                        bRouteest=false;
                                        bRoutereq=false;
                                        Returnmessage.iMessagetype =0; //Do nothing.
                                        bDirectroute =false;
                                        return(Returnmessage);
                                }
                        }
                        else
                        {
                                Returnmessage.iMessagetype =0; //Do nothing.
                                return(Returnmessage);
                        }
                }
 }
 else
 {
                // when the route is already established.
                if(iCurpacket < iNumberofpackets)
                {
                        //Send a Data packet.
                        iCurpacket++;
                        Returnmessage.iSourceid = iNodeid;
                        Returnmessage.iDestinationid = iCurdestin;
                        Returnmessage.iLocroute =1;
                        Returnmessage.iSenderid = iNodeid;
                        CRoutes Temproute;
                        if(iLastroute >= int((listRcollection.size()-1)))
                        {
                                Temproute = Return_route(0);
                                if(Temproute.iRouteid == -1)
                                {
                                        if((iTime - iLasttransfertime)> 15)
                                                {
                                                        bTransmitdata =false;
                                                        bTransferdata = false;
                                                        bRouteest =false;
                                                        bRoutereq =false;
                                                        bDirectroute =false;
                                                        iCurpacket =0;
                                                }
                                        Returnmessage.iMessagetype =0; //Do nothing.
                                        return(Returnmessage);
                                }
                                iLastroute=0;
                        }
```

```
					else
					{
						iLastroute++;
						Temproute= Return_route(iLastroute);
						if(Temproute.iRouteid == -1)
						{
							//Route is not found.
							Temproute = Return_route(0);
							if(Temproute.iRouteid == -1)
							{
								if((iTime - iLasttransfertime)> 15)
								{
									//Cancel transmission.
									bTransmitdata =false;
									bTransferdata = false;
									bRouteest =false;
									bRoutereq =false;
									bDirectroute =false;
									iCurpacket =0;
								}
								Returnmessage.iMessagetype =0; //Do nothing.
								return(Returnmessage);
							}
							iLastroute=0;
						}
					}

					Returnmessage.listRoute = Temproute.listRoute;
					Returnmessage.iRouteid = Temproute.iRouteid;
					Returnmessage.iReceiverid = Returnmessage.Returnrouteloc();
					if(Returnmessage.iReceiverid == -1)
					{
						Delete_route(Temproute.iRouteid, iNodeid);
						Returnmessage.iMessagetype =0;  //Do nothing.
						return(Returnmessage);
					}
					Returnmessage.iMessageid = iCurpacket;
					Returnmessage.iMessagetype =4; //Data Message.
					Returnmessage.iTimestamp = iTime;
					iPowerlevel = iPowerlevel - 250;
					lMessage_sent++;
					iLasttransfertime= iTime;
					if((iCurpacket == 10)&&(bDirectroute == false))
					{
						iNotransmisions++;
					}
					return(Returnmessage);
				}
				else
				{
					//When all the packets are transmitted.
					bTransmitdata =false;
					bRouteest =false;
					bRoutereq =false;
					bDirectroute = false;
					iCurpacket =0;
					Returnmessage.iMessagetype =0; //Do nothing.
					return(Returnmessage);
				}

			}

		}
		Returnmessage.iMessagetype =0;
		return(Returnmessage);
}
/*
		This function is one of the basic repsonsibilities of a node.
		Function is used to read the messages it has received. The messages
		are present in a queue and it reads them one at a time. After reading
		a message the node has to perform an action appropriately. The
		function is called until the list is empty or message time stamp is
```

```
		less than the current clock time.
*/
CMessage_sbcr CSBCRNode::Read_Message(long iTime)
{
		CMessage_sbcr Returnmessage;
		Returnmessage.iTimestamp = iTime;
		if(!listMessage.empty())
		{
				//Message list is found empty.
				if(iPowerlevel <=0)
				{
						//No power left.
						Returnmessage.iMessagetype = 6; //Node Dead.
						listMessage.clear();
						return(Returnmessage);
				}
				CMessage_sbcr Topmessage = listMessage.front();
				if(Topmessage.iTimestamp < iTime)
				{
						//Valid message to be read.
						listMessage.pop_front();
						if((iTime - Topmessage.iTimestamp) > 10)
						{
								//Message expired.
								iPowerlevel = iPowerlevel -5;
								Returnmessage.iMessagetype = 5; //Do nothing.
								return(Returnmessage);
						}
						if(Topmessage.iReceiverid == iNodeid)
						{
								//if the node is the intended receiver.
								if(Topmessage.iMessagetype==0)
								{
										iPowerlevel = iPowerlevel - 5;
										//Received a Beacon.
										int Tempsector;
										Tempsector = Sectorassociation(iCurrloc_X, iCurrloc_Y, Topmessage.iLocx,
Topmessage.iLocy);
										Add_adjacency(Topmessage.iSourceid,
Tempsector,Topmessage.iMessageid,iTime, Topmessage.iRouteid);
										Returnmessage.iMessagetype=5; //Do nothing.
										return(Returnmessage);
								}
								if(Topmessage.iMessagetype ==1)
								{
										//Requesting a Route.
										iPowerlevel = iPowerlevel - 50;

										if(Add_Rreqlist(Topmessage.iSourceid, Topmessage.iMessageid, iTime)==true)
										{
												if(Topmessage.iDestinationid==iNodeid)
												{
												//if the current node is the destination requested.
														Returnmessage.iMessagetype=2;// Start Route reply.
														Returnmessage.iDestinationid = Topmessage.iSourceid;
														Returnmessage.iNoofhops= Topmessage.iNoofhops;
														Returnmessage.iSourceid = iNodeid;
														Returnmessage.iSenderid = iNodeid;
														Returnmessage.Addtoroute(iNodeid);
														Returnmessage.iRouteid = 0; //Help decide the receivers.
														Returnmessage.iReceiverid = iNodeid; //Do not send to
same node.
														Returnmessage.iLocx = Topmessage.iLocx;
														Returnmessage.iLocy = Topmessage.iLocy;
														Returnmessage.iTimestamp = iTime;
														Returnmessage.iMessageid =0;
														Returnmessage.iLocroute =
Sectorassociation(iCurrloc_X, iCurrloc_Y, Topmessage.iLocx, Topmessage.iLocy);
														//RREP packet sent.
														return(Returnmessage);
												}
												else
```

```
				{
					//if an intermediate node then forward to destination.
					Returnmessage.Createrreq(Topmessage.iMessageid,
Topmessage.iDestinationid, Topmessage.iSourceid, Topmessage.iTimestamp, Topmessage.iLocx, Topmessage.iLocy);
					Returnmessage.iNoofhops= Topmessage.iNoofhops+1;
					Returnmessage.iSenderid= iNodeid;
					Returnmessage.iReceiverid = Topmessage.iSenderid;
//Do not send again to this node.
					iPowerlevel = iPowerlevel - 75;
					Returnmessage.iTimestamp = Topmessage.iTimestamp;
					lOverhead++;
					return(Returnmessage);
				}
			}
			else
			{
				Returnmessage.iMessagetype=5; //Ignore message;
				return(Returnmessage);
			}
		}
		if(Topmessage.iMessagetype ==2)
		{
			//Received a Route reply message.
			if(Topmessage.iDestinationid == iNodeid)
			{
				//RREP reaches the destination.
				if(Topmessage.iSourceid == iCurdestin)
				{
					//if it is valid RREP from the current destination.
					bRouteest=true;
					iNumberpaths++;
					CRoutes Temproute;
					Topmessage.Addtoroute(iNodeid);
					iLasttransfertime = iTime;
					bTransferdata = true;
					//Add the route.
					while(!Topmessage.listRoute.empty())
					{
						int iConst;
						unsigned int iSize;
						iSize = int(Topmessage.listRoute.size());
						iConst = Topmessage.listRoute[(iSize-1)];
						Temproute.listRoute.push_back(iConst);
						Topmessage.listRoute.pop_back();
					}
					Temproute.iRouteid = int(listRcollection.size());
					listRcollection.push_back(Temproute);
					Returnmessage.iMessagetype=5; //Do nothing.
					return(Returnmessage);
				}
				else
				{
					Returnmessage.iMessagetype=5; //Do nothing.
					return(Returnmessage);
				}
			}
			else
			{
				// Need to forward to another appropriate node.
				Returnmessage.iMessagetype=2;
				Returnmessage.iRouteid = 1;
				Returnmessage.listRoute = Topmessage.listRoute;
				Returnmessage.listRoute.push_back(iNodeid);
				if(Returnmessage.listRoute.size()>20)
				{
					Returnmessage.iMessagetype =5; //Do nothing.
					return(Returnmessage);
				}
				Returnmessage.iSourceid = Topmessage.iSourceid;
				Returnmessage.iDestinationid = Topmessage.iDestinationid;
				Returnmessage.iLocx = Topmessage.iLocx;
				Returnmessage.iLocy = Topmessage.iLocy;
```

```
						Returnmessage.iMessageid = Topmessage.iMessageid+1; //Has the
distance covered.
						Returnmessage.iSenderid = iNodeid;
						Returnmessage.iTimestamp = Topmessage.iTimestamp;
						Returnmessage.iNoofhops = Topmessage.iNoofhops; //Has the
actual distance.
						Returnmessage.iReceiverid = Decide_nextnode(iCurrloc_X,
iCurrloc_Y, Topmessage.iLocx, Topmessage.iLocy, Topmessage.iNoofhops,Returnmessage.iMessageid, Topmessage.iDestinationid,
Topmessage.iSenderid);
						if(Returnmessage.iReceiverid ==-1)
						{
							Returnmessage.iMessagetype =5; //Do nothing as node
not avialable.
						}
						else
						{
							//Do nothing.
						}
						return(Returnmessage);
					}
				}
				if(Topmessage.iMessagetype ==3)
				{
					//Received a Route reply error.
					if(Topmessage.iDestinationid == iNodeid)
					{
						//The RRER reaches the source of transmission.
						if((bTransmitdata ==true)&&(Topmessage.iMessageid ==
iCurdestin))
						{
							//iMessageid has the route destination stored in it.
							//Delete the route from the collection.
							Delete_route(Topmessage.iRouteid,
Topmessage.iSourceid);
							if(listRcollection.empty()==true)
							{
								if((iTime - iLasttransfertime) > 15)
								{
									//Cancel the transmission.
									bTransmitdata=false;
									bRouteest=false;
									bRoutereq =false;
									bDirectroute =false;
									iReqcount=0;
								}
							}
						}
						Returnmessage.iMessagetype =5; //Do nothing.
						return(Returnmessage);
					}
					else
					{
						//if an intermediate node has received the Error message.
						Returnmessage = Topmessage;
						Returnmessage.iSenderid = iNodeid;
						Returnmessage.iLocroute--;
						Returnmessage.iReceiverid = Returnmessage.Returnrouteloc();
		if(Returnmessage.iReceiverid == -1)
						{
							Returnmessage.iMessagetype = 5; //Do nothing.
							return(Returnmessage);
						}
						return(Returnmessage);
					}
				}
				if(Topmessage.iMessagetype ==4)
				{
					//Recevied a Data forward message.
					iPowerlevel = iPowerlevel - 150;
					if(Topmessage.iDestinationid == iNodeid)
					{
						//Recevied the Data packet.
```

```
        Returnmessage.iMessagetype=5; //Do nothing.
        lMessage_succes++;
        return(Returnmessage);
    }
    else
    {
        //The intermediate node in the path forwards message to next node.
        Returnmessage = Topmessage;
        Returnmessage.iSenderid = iNodeid;
        Returnmessage.iLocroute++;
        Returnmessage.iReceiverid = Returnmessage.Returnrouteloc();
        if(Returnmessage.iReceiverid == -1)
        {
            //Invalid reference found.
            lMessage_drop++;
            Returnmessage.iMessagetype =5; //Do nothing.
            return(Returnmessage);
        }
        if(Is_adjacent(Returnmessage.iReceiverid)==false)
        {
            //if next node is not in the adjacency.
            if(Add_Rerrorlist(Topmessage.iRouteid, Topmessage.iSourceid, Topmessage.iDestinationid, iTime)==false)
            {
                Returnmessage.iMessagetype =5; //Do nothing.
                lMessage_drop++;
                return(Returnmessage);
            }
            else
            {
                //RERR Message is issued.
                Returnmessage.iMessagetype =3;
                Returnmessage.iDestinationid = Topmessage.iSourceid;
                Returnmessage.iSourceid = iNodeid;
                Returnmessage.iLocroute= Returnmessage.iLocroute -2;
                Returnmessage.iReceiverid = Returnmessage.Returnrouteloc();
                if(Returnmessage.iReceiverid == -1)
                {
                    Returnmessage.iMessagetype = 5; //Do nothing.
                    return(Returnmessage);
                }

                iLasttransfertime = iTime;
                lMessage_drop++;
                return(Returnmessage);
            }

        }
        iLasttransfertime = iTime;
        iPowerlevel = iPowerlevel - 250;
        return(Returnmessage);
    }
}
if(Topmessage.iMessagetype==5)
{
    //Received an Invitation into a cluster.

    if(iCurstatus==1)
    {
        iNumberofinvitations++;
    }
    else
    {
        // if the node is a member or gateway.
        CMember Tempmember;
        Tempmember.iMemberid = Topmessage.iSourceid;
```

```
                                    Tempmember.iRole = Topmessage.iMessageid; //role is stored in
Mid.
                                    Tempmember.iSector = Sectorassociation(iCurrloc_X, iCurrloc_Y,
Topmessage.iLocx, Topmessage.iLocy);
                                    listMember.push_back(Tempmember);
                                    if(listMember.size()==1)
                                    {
                                        iCurstatus=2;
                                    }
                                    else
                                    {
                                        iCurstatus=3;
                                    }
                                }
                                Returnmessage.iMessagetype =5; //Do nothing.
                                return(Returnmessage);
                            }
                        }
                        else
                        {
                            Returnmessage.iMessagetype=5; //Ignore Message.
                            return(Returnmessage);
                        }
                    }
                    else
                    {
                        Returnmessage.iMessagetype =6; // No messages to read;
                        return(Returnmessage);
                    }
            }
            else
            {
                Returnmessage.iMessagetype =6; //No message to read;
                return(Returnmessage);
            }
            return(Returnmessage);
}
/*
            The function deletes a route from the collection of routes. The
            route ID is used to identify the route and is cross checked with
            a node present in the route.
*/
void CSBCRNode::Delete_route(int iId, int iSender)
{
            list <CRoutes>::iterator itRoutes;
            list <CRoutes> listTemp;
            bool bFlag, bFlag1;
            bFlag=false; bFlag1= false;
            //Locate route.
            for(itRoutes = listRcollection.begin(); itRoutes != listRcollection.end(); itRoutes++)
            {
                CRoutes Temproute = *itRoutes;
                if(Temproute.iRouteid == iId)
                {
                    //Route found.
                    unsigned int iTemp;
                    for(iTemp=0; iTemp < Temproute.listRoute.size(); iTemp++)
                    {
                        if(Temproute.listRoute[iTemp]== iSender)
                        {
                            //Check if sender node exist in it.
                            bFlag=true;
                            listRcollection.erase(itRoutes);
                            break;
                        }
                    }
                }
                //if the route was erased from list.
                if((bFlag==true) &&(bFlag1==true))
                {
                    Temproute.iRouteid--;
                    listTemp.push_back(Temproute);
```

```
				listRcollection.erase(itRoutes);
			}
			if(bFlag==true) { bFlag1=true; }
		}
		if(bFlag==true)
		{
			//Move from temp list back to route collection.
			while(!listTemp.empty())
			{
				listRcollection.push_back(listTemp.front());
				listTemp.pop_front();
			}
		}
}
/*
		The function is used to check if a node is a member of a cluster.
		Returns true or false.
*/
bool CSBCRNode::Is_member(int iId)
{
		list <CMember>::iterator itMember;
		for(itMember= listMember.begin(); itMember != listMember.end(); itMember++)
		{
			CMember Tempmember = *itMember;
			if(Tempmember.iMemberid == iId)
			{
				return(true);
			}
		}
		return(false);
}
/*
		The function decides which node should be selected during the
		route discovery process. it takes into consideration the
		current location of the node, the location of the sender (dest)
		the number of hops between source and destination, the distance
		covered, and the destination and source ids. A node is selected
		according to the SBCR route discovery process.
*/
int CSBCRNode::Decide_nextnode(int iL1x, int iL1y, int iL2x, int iL2y, int iDistance, int iDiscovered, int iDestinid, int iSenderid)
{
		list <CAdjacency_sbcr>::iterator itAdjacency;
		//if destination is present in the adjacency.
		for(itAdjacency= listAdjacency.begin(); itAdjacency!= listAdjacency.end(); itAdjacency++)
		{
			CAdjacency_sbcr Tempadj = *itAdjacency;
			if(Tempadj.iNeighborid == iDestinid)
			{
				return(iDestinid);
			}
		}
  int iSector;
  //Locate the sector direction.
  iSector = Sectorassociation(iL1x, iL1y, iL2x, iL2y);
  vector <int> listSectors;
  listSectors.push_back(iSector);
  //Decide the sectors from which a node should be considered.
  if(iDiscovered < (iDistance/2))
  {
		//if distance covered is less than half.
			if(iSectors < 7)
			{
				//Number of sectors are less than 7.
				if((iSector-1)<=0)
				{
					listSectors.push_back(iSectors);
				}
				else
				{
					listSectors.push_back((iSector-1));
				}
				if(((iSector+1)%(iSectors+1)==0))
```

```
                        {
                                listSectors.push_back(1);
                        }
                        else
                        {
                                listSectors.push_back((iSector+1));
                        }
                }
                else
                {
                        int iTemp;
                        for(iTemp=1; iTemp <=2; iTemp++)
                        {
                                if((iSector-iTemp)<=0)
                                {
                                        if((iSector-iTemp)<=-1)
                                        {
                                                listSectors.push_back(iSectors-1);
                                        }
                                        else
                                        {
                                                listSectors.push_back(iSectors);
                                        }
                                }
                                else
                                {
                                        listSectors.push_back((iSector-iTemp));
                                }
                                if(((iSector+iTemp)%(iSectors+1))==0)
                                {
                                        listSectors.push_back(1);
                                }
                                else
                                {
                                        listSectors.push_back((iSector+iTemp)%(iSectors+1));
                                }
                        }
                }
  }
  int iNodedecided, iMaxadjval;
  unsigned int iTemp;
  iNodedecided=-1; iMaxadjval=-1;
  list <CAdjacency_sbcr>::iterator itSbcradj;
  //Choose a node from the sectors.
  for(iTemp=0; iTemp < listSectors.size(); iTemp++)
  {
                for(itSbcradj=listAdjacency.begin(); itSbcradj!= listAdjacency.end(); itSbcradj++)
                {
                        CAdjacency_sbcr adjTemp = *itSbcradj;
                        if(adjTemp.iNeighborid != iSenderid)
                        {
                        if((adjTemp.iSector == listSectors[iTemp])&&(adjTemp.iAdjvalue > iMaxadjval))
                        {
                                if(((Is_member(adjTemp.iNeighborid)==true)&&(adjTemp.iRole==3))||(adjTemp.iRole
==1))
                                {
                                        //Choose a node that is a member and gateway node or cluster head.
                                    iNodedecided = adjTemp.iNeighborid;
                                        iMaxadjval = adjTemp.iAdjvalue;
                                }

                        }
                        }
                }
  }
  return(iNodedecided);
}
/*
        This function checks if a node has any nodes in one of its
        sectors. Returns true if they are present.
*/
bool CSBCRNode::Sector_hasnode(int iSecid)
```

```
{
	list <CAdjacency_sbcr>::iterator itAdj;
	//Check adjacency.
	for(itAdj=listAdjacency.begin(); itAdj!= listAdjacency.end(); itAdj++)
	{
		CAdjacency_sbcr Adjtemp =*itAdj;
		if(Adjtemp.iSector == iSecid)
		{
			return(true);
		}
	}
	return(false);
}
/*
	The function helps a node in calculating its Node distribution value.
	it checks the sectors of the nodes and the nodes present in them.
*/
int CSBCRNode::Weight(int iSec1, int iSec2)
{
	double dAngle1, dAngle2, dAngle3, dAngle4, dSectorangle;
	dSectorangle= double(6.284 / iSectors);
	dAngle1= double((iSec1-1)*dSectorangle);
	dAngle2= double(iSec1 * dSectorangle);
	dAngle3= double ((iSec2-1) *dSectorangle);
	dAngle4= double(iSec2 * dSectorangle);
	double dTempangle;
	dTempangle = (dAngle1+dAngle2)/2;
	dTempangle = dTempangle+3.142;
	if(dTempangle > 6.284)
	{
		dTempangle = dTempangle - 6.284;
	}
	if((dAngle3 <= dTempangle) && (dTempangle <= dAngle4))
	{
		return(iSectors/2); //Returns weight of opposite sector.
	}
	int iMaxval, iTemp;
	iMaxval = iSectors/2;
	double dBoundary1, dBoundary2;
	dTempangle = (dAngle1+dAngle2)/2;
	dBoundary1= dTempangle-1.571;
	dBoundary2= dTempangle+1.571;
	double dTempangle2;
	dTempangle2= dTempangle;
	//Check half of the sectors.
	for(iTemp=1; iTemp <= iMaxval; iTemp++)
	{
		dTempangle= dTempangle + 3.142+ double(dSectorangle * iTemp);
  if(dTempangle > 6.284)
		{
			dTempangle =dTempangle - 6.284;
		}
		if((dBoundary1 <= dTempangle)&&(dTempangle <= dBoundary2))
		{
			if((dAngle3 <= dTempangle)&&(dTempangle <= dAngle4))
			{
				return(iMaxval-iTemp);
				//Returns the weight associated with relative opposite sector.
			}
		}
	}
	return(0); //The sectors with no weight.
}
/*
	The function considers the node distribution value, adajcency value,
	security level, and power level to decide if the node should continue
	as a cluster head or should become a cluster head etc. if a node is
	involved in any data transmission then it with helds its decision.
*/
int CSBCRNode::Automataclusterdecision(long iTime, int iMaxsecuritylevel)
{
	int iNDValue, iWeight;
```

```
int iTemp;
iNDValue =0;
//Calculate the NDV.
for(iTemp=1; iTemp <= iSectors; iTemp++)
{
		if(Sector_hasnode(iTemp)==true)
		{
				iWeight=0;
				int iTemp1;
				for(iTemp1=1; iTemp1 <= iSectors; iTemp1++)
				{
						if((iTemp!= iTemp1)&&(Sector_hasnode(iTemp1)==true))
						{
								iWeight = iWeight + Weight(iTemp, iTemp1);
						}
				}
				iNDValue = iNDValue + iWeight;
		}
}
//Average ND value per sectors.
iNDValue = iNDValue / iSectors;
list <CAdjacency_sbcr>::iterator itAdj;
int iNohighsecurity;
//number of nodes with higher security level surrounding the curr node.
iNohighsecurity=0;
for(itAdj= listAdjacency.begin(); itAdj!= listAdjacency.end(); itAdj++)
{
		CAdjacency_sbcr Tempadj= *itAdj;
		if(Tempadj.iSecuritylevel > iSecuritylevel)
		{
				iNohighsecurity++;
		}
}
if(iCurstatus==0)
{
		//if node is undecided.
		if((iNDValue > (iSectors+1))&&(iPowerlevel > 1000))
		{
				//become a clusterhead.
				iCurstatus =1;
				return(0);
		}
		if((iNumberofinvitations > iNohighsecurity)&&(iSecuritylevel >= (iMaxsecuritylevel-2)))
		{
				//become a member.
				iCurstatus =2;
				return(1);
		}
		if(listAdjacency.size()==0)
		{
				//become a cluster head.
				iCurstatus =1;
		}
		return(0);
}
if(((iTime- iLasttransfertime)>10)||(iTime < 10))
{
		bTransferdata = false;
}
else
{
		//Invovled in transmission.
		bTransferdata =true;
}

if(iCurstatus>=1)
{
		//Node is currently a clusterhead, member or gateway node.
		if(bTransferdata==false)
		{
				if(iCurstatus ==1)
				{
```

```
						//Cluster head to undecided.
						if((int(listMember.size())< iNDValue)&&(iNumberofinvitations > iNohighsecurity))
						{
							iCurstatus =0;
							return(1);
						}

					}
					if(iCurstatus==2)
					{
						//Member to cluster head.
						if((iNDValue > (iSectors+1))&&(iPowerlevel > 100)&&(iNumberofinvitations <
iNohighsecurity))
						{
							iCurstatus =1;
							return(0);
						}
					}
					if(iCurstatus ==3)
					{
						//gateway  node to member.
						if(listMember.size()<=1)
						{
							iCurstatus =2;
							return(3);
						}
					}
				}
				if(iCurstatus ==2)
				{
					//Member to a gateway node.
					if(listMember.size() >1)
					{
						iCurstatus =3;
						return(3);
					}
				}
		}
		return(3);
}
/*
		Check if a node is present in the adjacency of the current
		node.
*/
bool CSBCRNode::Is_adjacent(int iId)
{
		list <CAdjacency_sbcr>::iterator itMember;
		for(itMember= listAdjacency.begin(); itMember != listAdjacency.end(); itMember++)
		{
				CAdjacency_sbcr Tempmember = *itMember;
				if(Tempmember.iNeighborid  == iId)
				{
					//Node found in adjacency.
					return(true);
				}
		}
		return(false);
}
/*
		The function removes the members that no longer are adjacent
		to the current node.
*/
void CSBCRNode::Update_member(void)
{
		list <CMember>::iterator itMember;
		list <CMember> Templist;
		for(itMember = listMember.begin(); itMember != listMember.end(); itMember++)
		{
				CMember Tempmember = *itMember;
				if(Is_adjacent(Tempmember.iMemberid)==true)
				{
					//Check the adjacency.
```

```
                        Templist.push_back(Tempmember);
                }
        }
        listMember.clear();
        //Move from temp list to member list.
        while(!Templist.empty())
        {
                itMember = Templist.begin();
                CMember Curr_node = *itMember;
          listMember.push_back(Curr_node);
                Templist.pop_front();
        }
}
//Default constructor of CSBCRNode class.
CSBCRNode::CSBCRNode(void)
{
        iSectors=0;
        iNodespersector=0;
        iCurdestin=0;
        iCurpacket=0;
        iCurstatus=0;
        bTransmitdata=false;
        bRoutereq=false;
        bRouteest=false;
        bTransferdata = false;
        iLasttransfertime=0;
        iRreqid=0;
        iReqcount=0;
        iInitialtime=0;
        iNumberofpackets=0;
        iNumberofinvitations=0;
        iLastroute=0;
        iNumberpaths=0;
        iNotransmisions=0;
        bDirectroute = false;
}
//Copy Constructor of CSBCRNode class.
CSBCRNode::CSBCRNode(const CSBCRNode &x1):CNode(x1)
{
        iSectors= x1.iSectors;
        iNodespersector= x1.iNodespersector;
        iCurdestin= x1.iCurdestin;
        iCurpacket=x1.iCurpacket;
        iCurstatus= x1.iCurstatus;
        bTransmitdata=x1.bTransmitdata;
        bRoutereq=x1.bRoutereq;
        bRouteest=x1.bRouteest;
        bTransferdata = x1.bTransferdata;
        iLasttransfertime= x1.iLasttransfertime;
        iRreqid= x1.iRreqid;
        iReqcount= x1.iReqcount;
        iInitialtime= x1.iInitialtime;
        iNumberofpackets= x1.iNumberofpackets;
        iNumberofinvitations= x1.iNumberofinvitations;
        iLastroute= x1.iLastroute;
        iNumberpaths = x1.iNumberpaths;
        iNotransmisions = x1.iNotransmisions;
        listGlobal = x1.listGlobal;
        listRcollection = x1.listRcollection;
        listMessage = x1.listMessage;
        listAdjacency = x1.listAdjacency;
        listMember = x1.listMember;
        listRreq = x1.listRreq;
        listRerr = x1.listRerr;
        bDirectroute = x1.bDirectroute;
}
//= Operator Overloading for CSBCRNode class.
CSBCRNode & CSBCRNode::operator =(CSBCRNode x1)
{
        iPowerlevel = x1.iPowerlevel;
        iSecuritylevel = x1.iSecuritylevel;
        iSpeed = x1.iSpeed;
```

```
		iCurrloc_X = x1.iCurrloc_X;
		iCurrloc_Y = x1.iCurrloc_Y;
		iFutloc_X = x1.iFutloc_X;
		iFutloc_Y = x1.iFutloc_Y;
		iBoundary_X = x1.iBoundary_X;
		iBoundary_Y = x1.iBoundary_Y;
		iMaxspeed = x1.iMaxspeed;
		bMobility = x1.bMobility;
		iBeacontime = x1.iBeacontime;
		iLastbeacon = x1.iLastbeacon;
		iNodeid = x1.iNodeid;
		iPausetime = x1.iPausetime;
		lMessage_succes = x1.lMessage_succes;
		lMessage_drop = x1.lMessage_drop;
		lOverhead = x1.lOverhead;
		lMessage_sent = x1.lMessage_sent;
		bCounted = x1.bCounted;
		iSectors= x1.iSectors;
		iNodespersector= x1.iNodespersector;
		iCurdestin= x1.iCurdestin;
		iCurpacket=x1.iCurpacket;
		iCurstatus= x1.iCurstatus;
		bTransmitdata=x1.bTransmitdata;
		bRoutereq=x1.bRoutereq;
		bRouteest=x1.bRouteest;
		bTransferdata = x1.bTransferdata;
		iLasttransfertime= x1.iLasttransfertime;
		iRreqid= x1.iRreqid;
		iReqcount= x1.iReqcount;
		iInitialtime= x1.iInitialtime;
		iNumberofpackets= x1.iNumberofpackets;
		iNumberofinvitations= x1.iNumberofinvitations;
		iLastroute= x1.iLastroute;
		iNumberpaths = x1.iNumberpaths;
		iNotransmisions = x1.iNotransmisions;
		listGlobal = x1.listGlobal;
		listRcollection = x1.listRcollection;
		listMessage = x1.listMessage;
		listAdjacency = x1.listAdjacency;
		listMember = x1.listMember;
		listRreq = x1.listRreq;
		listRerr = x1.listRerr;
		bDirectroute = x1.bDirectroute;
		return(*this);
}
/*
		The functions adds an entry to the Rerr list. if an entry is already
		present then it returns False, so the node will not send any RERR message
		into the network.
*/
bool CSBCRNode::Add_Rerrorlist(int Rid, int iSour, int iDestin, long iTime)
{
		list <CRerror>::iterator itPointer;
		bool bFlag;
		bFlag=true;

		//Check if entry present.
		for(itPointer = listRerr.begin(); itPointer != listRerr.end(); itPointer++)
		{
				CRerror Tempnode = *itPointer;
				if((Tempnode.iRouteid == Rid)&&(Tempnode.iSourceid == iSour)&&(Tempnode.iDestinationid == iDestin))
				{
						bFlag=false;
						break;
				}
		}
		if(bFlag==true)
		{
				//Add entry.
				CRerror Addnode;
				Addnode.iRouteid  = Rid;
				Addnode.iSourceid = iSour;
```

```
			Addnode.iDestinationid = iDestin;
			Addnode.iTimestamp = iTime;
			listRerr.push_back(Addnode);
		}
		return(bFlag);
}
/*
		The timeout entries are removed from the table.
*/
void CSBCRNode::Update_rerror(long iTime)
{
		list <CRerror>::iterator itPointer;
		list <CRerror> listTemp;
		while(!listRerr.empty())
		{
			itPointer= listRerr.begin();
			CRerror Curr_node = *itPointer;
			if((iTime-Curr_node.iTimestamp)<=30)
			{
				//Entry not timeout.
				listTemp.push_back(Curr_node);
			}
			listRerr.pop_front();
		}
		//Move from temp list to Rerr list.
		while(!listTemp.empty())
		{
			itPointer = listTemp.begin();
			CRerror Curr_node = *itPointer;
   listRerr.push_back(Curr_node);
			listTemp.pop_front();
		}
}
/*
		The function determines the nodes for which RREQ message
		should be forwarded. The node Ids are maintained in a list
		that is used in upper layer functions to simulate the
		actuall forwarding.
*/
void CSBCRNode::Forwardrreq(CMessage_sbcr Message)
{
		listGlobal.clear();
		list <CAdjacency_sbcr>::iterator itAdj;
		bool bNodeflag;
		bNodeflag = false;
		//Nodes from adjacency.
		for(itAdj = listAdjacency.begin(); itAdj != listAdjacency.end(); itAdj++)
		{
			CAdjacency_sbcr Tempadj = *itAdj;
			if(Message.iDestinationid == Tempadj.iNeighborid)
			{
				listGlobal.push_back(Tempadj.iNeighborid);
				bNodeflag = true;
				break;
			}
		}
		if(bNodeflag == false)
		{
			//if destination node not found.
			int iSec;
			int iCutoff;
			if(iSectors > 4)
			{
				iCutoff = 1;
			}
			else
			{
				iCutoff =2;
			}
			for(iSec=1; iSec <= iSectors; iSec++)
			{
				int iVal;
```

```
					iVal =0;
					//Choose nodes from adjacency.
					for(itAdj = listAdjacency.begin(); itAdj != listAdjacency.end(); itAdj++)
					{
						CAdjacency_sbcr Tempadj = *itAdj;
						if((Tempadj.iSector == iSec)&&(Tempadj.iNeighborid != Message.iReceiverid
)&&((Tempadj.iRole == 1)||(Tempadj.iRole == 3)))
						{
							//choose one node from each sector that is either a cluster head or gateway
node.
							iVal++;
							listGlobal.push_back(Tempadj.iNeighborid);
							if(iVal >= iCutoff)
							{
								break;
							}
						}
					}
				}
		}
}
/*
		The function is called by a cluster head to invite nodes present
		in its adjacency list. The function is called when the members
		are less than the maximum capacity.
*/
CMessage_sbcr CSBCRNode::Invitemember(void)
{
		int iSec;
		listGlobal.clear();
		CMessage_sbcr Returnmessage;
		Returnmessage.iMessagetype =0; //Do nothing.
		if(iCurstatus==1)
		{
				//if node is cluster head.
		Returnmessage.iSenderid = iNodeid;
		Returnmessage.iSourceid = iNodeid;
		Returnmessage.iMessageid = iCurstatus;
		Returnmessage.iLocx = iCurrloc_X;
		Returnmessage.iLocy = iCurrloc_Y;
		Returnmessage.iMessagetype = 5;
		//Check for each sector.
		for(iSec = 1; iSec <= iSectors; iSec++)
		{
				int iCount;
				iCount =0;
				list <CMember>::iterator itAdj;
				//Coun the members.
				for(itAdj=listMember.begin(); itAdj!= listMember.end(); itAdj++)
				{
					CMember Adjtemp =*itAdj;
					if(Adjtemp.iSector == iSec)
					{
						iCount++;
					}
				}
				//Nodes less than maximum number.
				while(iCount < iNodespersector)
				{
					int iSecuritylevel, iAdjvalue, iNid, iRole;
					iSecuritylevel =-1;
					iAdjvalue =-1;
					iNid=-1;
					list <CAdjacency_sbcr>::iterator itAdj;
					//Check nodes in adjacency.
					for(itAdj=listAdjacency.begin(); itAdj != listAdjacency.end(); itAdj++)
					{
	CAdjacency_sbcr Tempadj = *itAdj;
					 if(Tempadj.iSector == iSec)
					 {
						//Check the security value and adajcency value of the node.
	 int iTempvalue, iTempvalue1;
```

```
                                                iTempvalue = 2*Tempadj.iSecuritylevel + Tempadj.iAdjvalue;
                                                iTempvalue1= 2* iSecuritylevel + iAdjvalue;
                                                if((iTempvalue > iTempvalue1)&&(Is_member(Tempadj.iNeighborid)==false))
                                                {
                                                        //see that node was not previously a member.
                                                        iSecuritylevel = Tempadj.iSecuritylevel;
                                                        iAdjvalue = Tempadj.iAdjvalue;
                                                        iNid = Tempadj.iNeighborid;
                                                        iRole = Tempadj.iRole;
                                                }
                                        }
                                }
        if(iNid ==-1)
                                {
                                        break;
                                }
                                else
                                {
                                        if(iRole !=1)
                                        {
                                                Add_member(iNid, iSec, iRole, iSecuritylevel);
                                        }
                                        listGlobal.push_back(iNid);
                                        iCount++;
                                }
                        }
                }
        }

        return(Returnmessage);
}
/*
        This function is used to decided the nodes that will receive the
        RREP message whena RREQ message is received by the destination.
        Depending on the number of sectors the node receving RREP are
        decided. Input is the direction of source node.
*/
void CSBCRNode::Forwardinitialrrep(int iSectorto)
{
        listGlobal.clear();
        list <CAdjacency_sbcr>::iterator itAdj;
        int iSecnotto;
        if(iSectorto < (iSectors /2))
        {
                iSecnotto = (iSectors /2) + iSectorto;
        }
        else
        {
                iSecnotto = iSectors/2;
        }
        int iSec;
        iSec=1;
        int iTotalr;
        iTotalr=0;
        //Except the nodes in the opposite sector other are considered.
        while(iSec <= iSectors)
        {
                int iVal;
                iVal =0;
                for(itAdj = listAdjacency.begin(); itAdj != listAdjacency.end(); itAdj++)
                {
                        CAdjacency_sbcr Tempadj = *itAdj;
                        if((Tempadj.iSector != iSecnotto)&&(Tempadj.iSector == iSec)&&((Tempadj.iRole
==1)||(Tempadj.iRole ==3)))
                        {
                                listGlobal.push_back(Tempadj.iNeighborid);
                                iVal++;
                                if(iVal > (iSecuritylevel + 3))
                                {
                                        break;
                                }
                        }
```

```
			}
			iTotalr = iTotalr + iVal;
			if(iTotalr >= (3 * iSecuritylevel + iSectors))
			{
				//Limit on the number of nodes to receive the RREP.
				break;
			}
			iSec++;
		}
}
/*
		The functions calls other sub functions to update all the tables.
*/
void CSBCRNode::Updatealltables(long iTime)
{
		Update_adjacency(iTime);
		Update_Rreqlist(iTime);
		Update_member();
		Update_rerror(iTime);
}

/*
		File Name: SBCRSim.h
		The file contains the CSBCRsim class that simulates the Sector
		Based Clustering and Routing protocol. The functions belonging
		to this class are implemented in CSBCRSim.cpp file.
*/
# pragma once
# include "SBCRNode.h"
# include <math.h>
# include <vector>
# include <iostream>
# include <fstream>
using namespace std;
/*
		The CSBCRsim class has the variables and functions to simulate
		Sector Based Clustering and Routing. Additional members are
		present that collect the simulation results.
*/
class CSBCRSim
{
public:
		vector <CSBCRNode> Nodelist; //Node list.
		vector <int> listGlobal3; //Temprorary list.
		long lClock; //Simulator clock.
		int iSectorsincluster; //Sectors present for a node.
		int iNodespersector; //Nodes per sector as member.
		int iNonodesdied; //Number of nodes dead.
		int iRadius; //Radius of communication.
		int iMaxnodes; //Maximum nodes in network.
		int iMaxdesiredspeed; //Speed of a node.
		int iBoundaryX; //Boundary X-coordinate.
		int iBoundaryY; //Boundary Y-coordinate.
		int iMaxsecurity; //Maximum security for a node.
		long iMaxpowerlevel; //Maximum power level.
		//Simulation results collection.
		long lTotal_message_succesful; //Messages successfull.
		long lTotal_overhead; //Overhead in network.
		long lTotal_message_dropped; //Messages dropped by nodes.
		long lTotal_message_sent; //Messages sent.
		long lTotal_power; //Total power of nodes in network.
		long lTotalpaths; //Number of paths used.
		long lTotaltransmisions; //Number of transmissions.
		long lTotal_transattempts; //Total transmission attempts.
		long lTotal_message_lost; //Messages lost in network.
		//Functions.
		CSBCRSim(void);
		void Send_beacon_to_nodes(int iId);
		void Sendmessage(CMessage_sbcr Message);
		void Sendrreq(CMessage_sbcr Message);
		void Sendinvitations(CMessage_sbcr Message);
		void SBCRsimulation(ostream &out, unsigned int lSeed, double dPercent);
```

```
        void SBCRsimtime(ostream &out, unsigned int lSeed, long lMaxc);
};

/*
        File Name: SBCRSim.cpp
        The file has implementations of functions that are members
        of CSBCRSim class present in SBCRSim.h file.
*/
# include "SBCRSim.h"
//Default constructor of CSBCRSim class.
CSBCRSim::CSBCRSim(void)
{
        lClock=0;
        iNonodesdied=0;
        iRadius=0;
        iMaxnodes=0;
        iMaxdesiredspeed=0;
        iBoundaryX=0;
        iBoundaryY=0;
        iMaxpowerlevel=0;
        iMaxsecurity =0;
        lTotal_message_succesful=0;
        lTotal_overhead=0;
        lTotal_message_dropped=0;
        lTotal_message_sent=0;
        lTotal_power=0;
        iSectorsincluster=0;
        iNodespersector=0;
        lTotalpaths =0;
        lTotaltransmisions =0;
        lTotal_transattempts=0;
        lTotal_message_lost=0;
}
/*
        The function is used to send beacons between nodes. It has
        a node ID as input. It sends beacons to all the nodes that
        are physically surrounding the input node.
*/
void CSBCRSim::Send_beacon_to_nodes(int iId)
{
        unsigned int iTemp;
        //Locate the node in the list.
        for(iTemp=0; iTemp < Nodelist.size(); iTemp++)
        {
                if(Nodelist[iTemp].iNodeid == iId)
                {
                        break;
                }
        }
        //Create beacon.
        CMessage_sbcr Newmessage;
        Newmessage.iSourceid = iId;
        Newmessage.iMessagetype = 0;
        Newmessage.iSenderid = iId;
        Newmessage.iMessageid = Nodelist[iTemp].iCurstatus;
        Newmessage.iRouteid = Nodelist[iTemp].iSecuritylevel;
        Newmessage.iTimestamp  = lClock;
        Nodelist[iTemp].iLastbeacon=lClock;
        int xother, yother, xthis, ythis;
  double Caldistance;
  //Get the current location of the node.
  xthis = Nodelist[iTemp].iCurrloc_X;
  ythis = Nodelist[iTemp].iCurrloc_Y;
  unsigned int iTemp1;
  //For each node in the network.
  for(iTemp1=0; iTemp1 <Nodelist.size(); iTemp1++)
  {
          if(iTemp != iTemp1)
          {
                  //Calculate the distance from the node sending beacon.
                  xother = Nodelist[iTemp1].iCurrloc_X;
                  yother = Nodelist[iTemp1].iCurrloc_Y;
```

```
                        Caldistance=sqrt(double((xother-xthis)*(xother-xthis) + (yother-ythis)*(yother-ythis)));
                        if(iRadius >= int(Caldistance))
                        {
                                    //if node in range send beacon.
                                    Newmessage.iDestinationid = iTemp1;
                                    Newmessage.iReceiverid  = iTemp1;
                                    Newmessage.iLocx = Nodelist[iTemp1].iCurrloc_X;
                                    Newmessage.iLocy = Nodelist[iTemp1].iCurrloc_Y;
                                    Nodelist[iTemp1].listMessage.push_back(Newmessage);
                        }
            }
  }
}
/*
            The function is used to send the input message between
            two nodes. The sender and receiver nodes are identified
            from the message. if the nodes are out of reach then
            the message is ignored.
*/
void CSBCRSim::Sendmessage(CMessage_sbcr Message)
{
 int xother, yother, xthis, ythis;
 double Caldistance;
 int iTonodeid, iSendnodeid;
 iTonodeid = Message.iReceiverid; //Receiver node.
 iSendnodeid = Message.iSenderid; //Sender node.
 unsigned int iTemp;
 //Locate sender node.
 for(iTemp = 0; iTemp < Nodelist.size(); iTemp ++)
 {
            if(iSendnodeid == Nodelist[iTemp].iNodeid)
            {
                        break;
            }
 }
 //Get sender current position.
  xthis = Nodelist[iTemp].iCurrloc_X;
  ythis = Nodelist[iTemp].iCurrloc_Y;
  //Locate Receiver.
 for(iTemp = 0; iTemp < Nodelist.size(); iTemp ++)
 {
            if(iTonodeid == Nodelist[iTemp].iNodeid)
            {
                        break;
            }
 }
 //Get Receiver current position.
 xother = Nodelist[iTemp].iCurrloc_X;
 yother = Nodelist[iTemp].iCurrloc_Y;
 //Calculate the distance between them.
 Caldistance=sqrt(double((xother-xthis)*(xother-xthis) + (yother-ythis)*(yother-ythis)));
 if(int(Caldistance) < iRadius)
 {
            //Send message.
            Nodelist[iTemp].listMessage.push_back(Message);
 }
 else
 {
            //Ignore message.
            lTotal_message_lost++;
 }

}
/*
            The function sends RREQ message to all the node present
            in the temporary list. The input message is forwarded to a
            node using the Sendmessage function.
*/
void CSBCRSim::Sendrreq(CMessage_sbcr Message)
{
 unsigned int iTemp;
 for(iTemp=0; iTemp < listGlobal3.size(); iTemp++)
```

```
 {
          Message.iReceiverid  = listGlobal3[iTemp];
          Sendmessage(Message);
 }
 listGlobal3.clear();
}
/*
         The function sends invitation message to all the node present
         in the temporary list. The input message is forwarded to a
         node using the Sendmessage function.
*/
void CSBCRSim::Sendinvitations(CMessage_sbcr Message)
{
 unsigned int iTemp;
 for(iTemp=0; iTemp < listGlobal3.size(); iTemp++)
 {
          Message.iReceiverid  = listGlobal3[iTemp];
          Sendmessage(Message);
 }
 listGlobal3.clear();
}
/*
         The SBCRsimulation function simulates SBCR protocol until the
         input percentage of nodes are dead. The function has three inputs
         output file stream, random function seed value and percentage of
         node in network to be dead.
*/
void CSBCRSim::SBCRsimulation(ostream &out, unsigned int lSeed, double dPercent)
{
         //Initially assign the desired number of nodes with the given parameters.
         unsigned int iNodes;
         srand(lSeed);
         for(iNodes=0; int(iNodes) < iMaxnodes; iNodes++)
         {
                  CSBCRNode Tempnode;
                  Tempnode.iNodeid= iNodes;
                  Tempnode.iMaxspeed = iMaxdesiredspeed;
                  Tempnode.iBoundary_X = iBoundaryX;
                  Tempnode.iBoundary_Y = iBoundaryY;
                  Tempnode.iCurrloc_X = rand()%iBoundaryX;
                  Tempnode.iCurrloc_Y = rand()%iBoundaryY;
                  Tempnode.iPowerlevel = iMaxpowerlevel;
                  Tempnode.iSectors = iSectorsincluster;
                  Tempnode.iNodespersector = iNodespersector;
                  Tempnode.iSecuritylevel = rand()%iMaxsecurity;
    Nodelist.push_back(Tempnode);
         }
         lClock=0;
         while (iNonodesdied < (iMaxnodes * (dPercent/100)))
         {
                  //Untill the constraint of dead nodes is satisfied.
                  lClock++;
                  //For each node in the network.
                  for(iNodes=0; iNodes < Nodelist.size(); iNodes++)
                  {
                           if((Nodelist[iNodes].iPowerlevel <=0)&&(Nodelist[iNodes].bCounted == false))
                           {
                                    //if the dead node was not considered previously.
                                    Nodelist[iNodes].bCounted = true;
                                    iNonodesdied++;
                           }
                  if(Nodelist[iNodes].iPowerlevel >0)
                  {
                           if(lClock > 5)
                           {
                                    //To give time for the network to settle down.
                                    Nodelist[iNodes].Set_future();
                                    Nodelist[iNodes].Move_node();
                                    Nodelist[iNodes].Determine_Send_Message(lClock, int(Nodelist.size()));
                                    CMessage_sbcr Tempmessage = Nodelist[iNodes].Send_Message(lClock);
                                    if(Tempmessage.iMessagetype != 0)
                                    {
```

```
				if(Tempmessage.iMessagetype ==1)
				{
					//RREQ packet to be forwarded.
					Nodelist[iNodes].Forwardrreq(Tempmessage);
					unsigned int iVec;
					listGlobal3.clear();

					for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
					{
						listGlobal3.push_back(Nodelist[iNodes].listGlobal[iVec]);
					}
					Sendrreq(Tempmessage);
				}
				else
				{
					//Send message.
					Sendmessage(Tempmessage);
				}
			}
		}
		bool bBeacon;
		// Decide whether to send a beacon.
		bBeacon=Nodelist[iNodes].Send_beacon(lClock);
		if(bBeacon== true)
		{
			//Send beacon.
			Send_beacon_to_nodes(iNodes);
		}
		CMessage_sbcr Tempmessage;
		try
		{
		while(1)
		{
			//Read message.
			Tempmessage = Nodelist[iNodes].Read_Message(lClock);

			if(Tempmessage.iMessagetype == 6)
			{
				//No more message to read.
				break;
			}
			if(Tempmessage.iMessagetype == 1)
			{
				//Send a RREQ packet.
				Nodelist[iNodes].Forwardrreq(Tempmessage);
				unsigned int iVec;
				listGlobal3.clear();
				for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
				{
					listGlobal3.push_back(Nodelist[iNodes].listGlobal[iVec]);
				}
				Sendrreq(Tempmessage);
			}
			if((Tempmessage.iMessagetype ==2)&&(Tempmessage.iRouteid== 0))
			{
				//Send initial RREP packets to nodes.
				Nodelist[iNodes].Forwardinitialrrep(Tempmessage.iLocroute);
				unsigned int iVec;
				listGlobal3.clear();
				for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
				{
					listGlobal3.push_back(Nodelist[iNodes].listGlobal[iVec]);
				}
				Sendrreq(Tempmessage);
			}
			else
			{
				if(Tempmessage.iMessagetype != 5)
				{
					//Send messages.
					Sendmessage(Tempmessage);
				}
```

```
                                                }
                                        }
                                        //Update tables and maintain cluster consistency.
                                        Nodelist[iNodes].Updatealltables(lClock);
                                        int iRetval;
        iRetval=Nodelist[iNodes].Automataclusterdecision(lClock, iMaxsecurity);
                                        if(iRetval==1)
                                        {
                                                Send_beacon_to_nodes(iNodes);
                                        }
                                        CMessage_sbcr Returnmessage;
                                        Returnmessage = Nodelist[iNodes].Invitemember();
                                        if(Returnmessage.iMessagetype !=0)
                                        {
                                                //Send invitations to nodes.
                                                unsigned int iVec;
                                                listGlobal3.clear();
                                                for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
                                                        {
                                                                listGlobal3.push_back(Nodelist[iNodes].listGlobal[iVec]);
                                                        }
                                                Sendinvitations(Returnmessage);
                                        }
                                }

                                catch(exception &e)
                                        {
                                                out<<"Exception occurred over here"<<e.what()<<endl;
                                                exit(0);
                                        }
                                }
                                else
                                {
                                        Nodelist[iNodes].listMessage.clear();
                                }
                                }
                }
                //Simulation results collection.
    for(iNodes=0; iNodes < Nodelist.size(); iNodes++)
    {
                lTotal_message_succesful= lTotal_message_succesful + Nodelist[iNodes].lMessage_succes;
                lTotal_overhead= lTotal_overhead + Nodelist[iNodes].lOverhead;
                lTotal_message_dropped= lTotal_message_dropped + Nodelist[iNodes].lMessage_drop;
                lTotal_message_sent= lTotal_message_sent + Nodelist[iNodes].lMessage_sent;
                lTotal_power= lTotal_power + Nodelist[iNodes].iPowerlevel;
                lTotalpaths = lTotalpaths + Nodelist[iNodes].iNumberpaths;
                lTotaltransmisions = lTotaltransmisions + Nodelist[iNodes].iNotransmisions;
                lTotal_transattempts = lTotal_transattempts + Nodelist[iNodes].lTotaltrans;
    }
    //Output results.
    out<<" Sector Based Clustering & Routing Protocol Simulation"<<endl;
    out<<"\n";
    out<<"Number of Nodes: "<<iMaxnodes<<"\n";
    out<<"Simulation Time: "<<lClock<<"\n";
    out<<"Number of Nodes Dead: "<<iNonodesdied<<"\n";
    out<<"Average Power Left of a Node: "<<double(lTotal_power / iMaxnodes)<<"\n";
    out<<"Number of Message Sent: "<<lTotal_message_sent<<"\n";
    out<<"Number of Message reached Destination: "<<lTotal_message_succesful<<"\n";
    out<<"Number of Message dropped: "<<lTotal_message_dropped<<"\n";
    out<<"Total Overhead: "<<lTotal_overhead<<"\n";
    out<<"Total Transmission attempts: "<<lTotal_transattempts<<"\n";
    out<<"% of Messages transmited succesufuly: "<<((double (lTotal_message_succesful)/double(lTotal_message_sent))*100)<<"\n";
    out<<"Total Number of paths: "<<lTotalpaths<<"\n";
    out<<"Total Transmisions: "<<lTotaltransmisions<<"\n";
    out<<"Total Message lost in Network: "<<lTotal_message_lost<<"\n";
    out<<"Average number of paths provided: "<<(double(lTotalpaths)/double(lTotaltransmisions))<<"\n"<<endl;
    out<<"\n";
}
/*
                The SBCRsimtime funciton is simular to the above function with the
                difference in the constraint. The maximum clock value is the constraint
                overhere. The simulation is performed until that clock value is reached.
```

```
*/
void CSBCRSim::SBCRsimtime(ostream &out, unsigned int lSeed, long lMaxc)
{
//Initially assign the desired number of nodes with the given parameters.
		unsigned int iNodes;
		srand(lSeed);
		for(iNodes=0; int(iNodes) < iMaxnodes; iNodes++)
		{
			CSBCRNode Tempnode;
			Tempnode.iNodeid= iNodes;
			Tempnode.iMaxspeed = iMaxdesiredspeed;
			Tempnode.iBoundary_X = iBoundaryX;
			Tempnode.iBoundary_Y = iBoundaryY;
			Tempnode.iCurrloc_X = rand()%iBoundaryX;
			Tempnode.iCurrloc_Y = rand()%iBoundaryY;
			Tempnode.iPowerlevel = iMaxpowerlevel;
			Tempnode.iSectors = iSectorsincluster;
			Tempnode.iNodespersector = iNodespersector;
			Tempnode.iSecuritylevel = rand()%iMaxsecurity;
	Nodelist.push_back(Tempnode);
		}
		lClock=0;
		while (lClock < lMaxc)
		{
			//Clock value constraint.
			lClock++;
			//For each node in the list.
			for(iNodes=0; iNodes < Nodelist.size(); iNodes++)
			{
				if((Nodelist[iNodes].iPowerlevel <=0)&&(Nodelist[iNodes].bCounted == false))
				{
					//Node dead was not previously considered.
					Nodelist[iNodes].bCounted = true;
					iNonodesdied++;
				}
			if(Nodelist[iNodes].iPowerlevel >0)
			{
				//if node does not have any power resources.
				if(lClock > 5)
				{
					//To give time for the network to settle down.
					Nodelist[iNodes].Set_future();
					Nodelist[iNodes].Move_node();
					Nodelist[iNodes].Determine_Send_Message(lClock, int(Nodelist.size()));
					CMessage_sbcr Tempmessage = Nodelist[iNodes].Send_Message(lClock);
					if(Tempmessage.iMessagetype != 0)
					{
						if(Tempmessage.iMessagetype ==1)
						{
							//Send RREQ packet.
							Nodelist[iNodes].Forwardrreq(Tempmessage);
							unsigned int iVec;
							listGlobal3.clear();

							for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
							{
								listGlobal3.push_back(Nodelist[iNodes].listGlobal[iVec]);
							}
							Sendrreq(Tempmessage);
						}
						else
						{
							//send message.
							Sendmessage(Tempmessage);
						}
					}
				}
				bool bBeacon;
				//Decide whether to send beacon.
				bBeacon=Nodelist[iNodes].Send_beacon(lClock);
				if(bBeacon== true)
				{
```

```
				//send beacon.
				Send_beacon_to_nodes(iNodes);
			}
			CMessage_sbcr Tempmessage;
			try
			{
			while(1)
			{
				//Read message.
				Tempmessage = Nodelist[iNodes].Read_Message(lClock);

				if(Tempmessage.iMessagetype == 6)
				{
					//No more messages to read.
					break;
				}
				if(Tempmessage.iMessagetype == 1)
				{
					//Send RREQ packet.
					Nodelist[iNodes].Forwardrreq(Tempmessage);
					unsigned int iVec;
					listGlobal3.clear();
					for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
					{
						listGlobal3.push_back(Nodelist[iNodes].listGlobal[iVec]);
					}
					Sendrreq(Tempmessage);
				}
				if((Tempmessage.iMessagetype ==2)&&(Tempmessage.iRouteid== 0))
				{
					//Send initial RREP packet.
					Nodelist[iNodes].Forwardinitialrrep(Tempmessage.iLocroute);
					unsigned int iVec;
					listGlobal3.clear();
					for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
					{
						listGlobal3.push_back(Nodelist[iNodes].listGlobal[iVec]);
					}
					Sendrreq(Tempmessage);
				}
				else
				{
					if(Tempmessage.iMessagetype != 5)
					{
						//Send message.
						Sendmessage(Tempmessage);
					}
				}
			}
			//Update tables and check cluster consistency.
			Nodelist[iNodes].Updatealltables(lClock);
			int iRetval;
iRetval=Nodelist[iNodes].Automataclusterdecision(lClock, iMaxsecurity);
			if(iRetval==1)
			{
				Send_beacon_to_nodes(iNodes);
			}
			CMessage_sbcr Returnmessage;
			Returnmessage = Nodelist[iNodes].Invitemember();
			if(Returnmessage.iMessagetype !=0)
			{
				//Send invitations.
				unsigned int iVec;
				listGlobal3.clear();
				for(iVec=0; iVec < Nodelist[iNodes].listGlobal.size(); iVec++)
					{
						listGlobal3.push_back(Nodelist[iNodes].listGlobal[iVec]);
					}
				Sendinvitations(Returnmessage);
			}
		}
```

```
			catch(exception &e)
				{
					out<<"Exception occurred over here"<<e.what()<<endl;
					exit(0);
				}
			}
			else
			{
				Nodelist[iNodes].listMessage.clear();
			}
			}
		}
		//Simulation results collection.
  for(iNodes=0; iNodes < Nodelist.size(); iNodes++)
  {
		lTotal_message_succesful= lTotal_message_succesful + Nodelist[iNodes].lMessage_succes;
		lTotal_overhead= lTotal_overhead + Nodelist[iNodes].lOverhead;
		lTotal_message_dropped= lTotal_message_dropped + Nodelist[iNodes].lMessage_drop;
		lTotal_message_sent= lTotal_message_sent + Nodelist[iNodes].lMessage_sent;
		lTotal_power= lTotal_power + Nodelist[iNodes].iPowerlevel;
		lTotalpaths = lTotalpaths + Nodelist[iNodes].iNumberpaths;
		lTotaltransmisions = lTotaltransmisions + Nodelist[iNodes].iNotransmisions;
		lTotal_transattempts = lTotal_transattempts + Nodelist[iNodes].lTotaltrans;
  }
  //Output results to file.
  out<<" Sector Based Clustering & Routing Protocol Simulation"<<endl;
  out<<"\n";
  out<<"Number of Nodes: "<<iMaxnodes<<"\n";
  out<<"Simulation Time: "<<lClock<<"\n";
  out<<"Number of Nodes Dead: "<<iNonodesdied<<"\n";
  out<<"Average Power Left of a Node: "<<double(lTotal_power / iMaxnodes)<<"\n";
  out<<"Number of Message Sent: "<<lTotal_message_sent<<"\n";
  out<<"Number of Message reached Destination: "<<lTotal_message_succesful<<"\n";
  out<<"Number of Message dropped: "<<lTotal_message_dropped<<"\n";
  out<<"Total Overhead: "<<lTotal_overhead<<"\n";
  out<<"Total Transmission attempts: "<<lTotal_transattempts<<"\n";
  out<<"% of Messages transmited succesufuly: "<<((double (lTotal_message_succesful)/double(lTotal_message_sent))*100)<<"\n";
  out<<"Total Number of paths: "<<lTotalpaths<<"\n";
  out<<"Total Transmisions: "<<lTotaltransmisions<<"\n";
  out<<"Total Message lost in Network: "<<lTotal_message_lost<<"\n";
  out<<"Average number of paths provided: "<<(double(lTotalpaths)/double(lTotaltransmisions))<<"\n"<<endl;
  out<<"\n";
}

/*
		File Name: Mainsim.cpp
		This file is the starting point of simulation. The user
		options are considered here and appropriate simulation is
		started.
*/
# include <iostream>
# include <fstream>
# include "AodvSim.h"
# include "CBRSim.h"
# include "SBCRSim.h"
using namespace std;
# include <time.h>

/*
		The main function is the user interface that provides the user
		with options to select the type of simulation he wants. Also
		the user is asked to set the simulation parameters. The inputs
		like number of nodes, boundary coordinates, maximum security level,
		nodes communication radius, number of sectors, nodes per sector,
		maximum speed, maximum power level, and output file name are entered
		here. Depending on the simulation selected the % of nodes to be
		dead is requested for Type-1 simulation and the maximum clock value
		for Type-2 and Type-3 simulation. The current system time is used
		to set the seed value that is used for simulating all the three
		protocols.
*/
 int main()
```

```
{
        int iNonodes, iBoud_x, iBoud_y,iSecurity, iRadius, iNosectors, iNodepersec, iMaxspeed, iOption;
        long lMaxpower, lMaxclock;
        double dPernode;
        unsigned int seed;
         //Accept input values.
        cout<<" Input the number of nodes: "<<endl;
        cin>>iNonodes;
        cout<<"Input the Boudary of plain: "<<endl;
        cin>>iBoud_x>>iBoud_y;
        cout<<"Input Maximum security level desired: "<<endl;
        cin>>iSecurity;
        cout<<"Input Nodes Communication radius: "<<endl;
        cin>>iRadius;
        cout<<"Input Number of Sectors and Nodes per Sector: "<<endl;
        cin>>iNosectors>>iNodepersec;
        cout<<"The Maxspeed of nodes and Max power associated with them: "<<endl;
        cin>>iMaxspeed>>lMaxpower;
        cout<<"Output file name: "<<endl;
        char filename[100];
        cin>>filename;
        ofstream out;
        out.open(filename,ios::out);
        //Calculate the seed value.
        tm *starttime;
  time_t rawtime;
        time(&rawtime);
        starttime= localtime(&rawtime);
  seed = int(((starttime->tm_sec) + (2* starttime->tm_min) + (3* starttime->tm_hour)));
        //Take the user options for simulation.
        cout<<"Enter the option 1- Nodes dead 2- Time period 3- Sector Evaluation: ";
  cin>>iOption;
        if(iOption ==2)
        {
                //Type-2 simulation with time constraint.
                cout<<"Enter the Max Time period the simulations should run for: ";
                cin>>lMaxclock;
                CAodvSim Aodv;
                //Simulating Ad hoc On-demand Distance Vector.
                Aodv.iMaxnodes = iNonodes;
                Aodv.iBoundaryX = iBoud_x;
                Aodv.iBoundaryY = iBoud_y;
                Aodv.iMaxsecurity = iSecurity;
                Aodv.iRadius = iRadius;
                Aodv.iMaxdesiredspeed = iMaxspeed;
                Aodv.iMaxpowerlevel = lMaxpower;
                Aodv.Aodvsimtime(out, seed, lMaxclock);
                //Simulating Cluster Based Routing.
                CCBRSim Cbr;
                Cbr.iMaxnodes = iNonodes;
                Cbr.iBoundaryX = iBoud_x;
                Cbr.iBoundaryY = iBoud_y;
                Cbr.iMaxdesiredspeed = iMaxspeed;
                Cbr.iMaxsecurity = iSecurity;
                Cbr.iRadius = iRadius;
                Cbr.iMaxpowerlevel = lMaxpower;
                Cbr.CBRsimtime(out, seed, lMaxclock);
                //Simulating Sector based Clustering and Routing.
                CSBCRSim Sbcr;
                Sbcr.iMaxnodes = iNonodes;
                Sbcr.iBoundaryX = iBoud_x;
                Sbcr.iBoundaryY = iBoud_y;
                Sbcr.iMaxdesiredspeed = iMaxspeed;
                Sbcr.iMaxpowerlevel = lMaxpower;
                Sbcr.iMaxsecurity = iSecurity;
                Sbcr.iRadius = iRadius;
                Sbcr.iSectorsincluster = iNosectors;
                Sbcr.iNodespersector = iNodepersec;
                Sbcr.SBCRsimtime(out, seed, lMaxclock);

        }
        else
```

```
	{
		if(iOption ==3)
		{
			//Type-3 Simulation ony SBCR varying sectors in cluster.
			cout<<"Enter the Max Time period the simulations should run for: ";
			cin>>lMaxclock;
			CSBCRSim Sbcr;
			//Simulating SBCR.
			Sbcr.iMaxnodes = iNonodes;
			Sbcr.iBoundaryX = iBoud_x;
			Sbcr.iBoundaryY = iBoud_y;
			Sbcr.iMaxdesiredspeed = iMaxspeed;
			Sbcr.iMaxpowerlevel = lMaxpower;
			Sbcr.iMaxsecurity = iSecurity;
			Sbcr.iRadius = iRadius;
			Sbcr.iSectorsincluster = iNosectors;
			Sbcr.iNodespersector = iNodepersec;
			Sbcr.SBCRsimtime(out, seed, lMaxclock);

		}
		else
		{
			//Type-1 Simulation with number of nodes dead as constriant.
			cout<<"Enter What % of nodes should be dead: "<<endl;
			cin>>dPernode;
			CAodvSim Aodv;
			//Simulating AODV protocol.
			Aodv.iMaxnodes = iNonodes;
			Aodv.iBoundaryX = iBoud_x;
			Aodv.iBoundaryY = iBoud_y;
			Aodv.iMaxsecurity = iSecurity;
			Aodv.iRadius = iRadius;
			Aodv.iMaxdesiredspeed = iMaxspeed;
			Aodv.iMaxpowerlevel = lMaxpower;
			Aodv.Aodvsimulation(out,seed, dPernode);
			//Simulating Cluster Based Routing.
			CCBRSim Cbr;
			Cbr.iMaxnodes = iNonodes;
			Cbr.iBoundaryX = iBoud_x;
			Cbr.iBoundaryY = iBoud_y;
			Cbr.iMaxdesiredspeed = iMaxspeed;
			Cbr.iMaxsecurity = iSecurity;
			Cbr.iRadius = iRadius;
			Cbr.iMaxpowerlevel = lMaxpower;
			Cbr.CBRsimulation(out, seed, dPernode);
			//Simulating Sector based Clustering and Routing.
			CSBCRSim Sbcr;
			Sbcr.iMaxnodes = iNonodes;
			Sbcr.iBoundaryX = iBoud_x;
			Sbcr.iBoundaryY = iBoud_y;
			Sbcr.iMaxdesiredspeed = iMaxspeed;
			Sbcr.iMaxpowerlevel = lMaxpower;
			Sbcr.iMaxsecurity = iSecurity;
			Sbcr.iRadius = iRadius;
			Sbcr.iSectorsincluster = iNosectors;
			Sbcr.iNodespersector = iNodepersec;
			Sbcr.SBCRsimulation(out, seed, dPernode);
		}
	}
  out.close();
  return(1);
}
```

VITA

Sudheer Krishna Chimbli Venkata

Candidate for the Degree of

Master of Science

Thesis: SECTOR BASED CLUSTERING & ROUTING

Major Field: Computer Science

Biographical:

Personal Data: Born in Hyderabad, INDIA, June 24, 1981, son of Mr. Balakrishna Naidu Chimbli and Mrs. Rajyalakshmi T.R. Chimbli.

Education: Received the degree of Bachelor of Technology in Computer Science and Engineering from Jawaharlal Nehru Technological University, Hyderabad, India, in May 2002; completed the requirements for the Master of Science degree at the Computer Science Department at Oklahoma State University, Stillwater, Oklahoma, in December 2004.

Experience: Research Assistant in the Department of Computer Science, Oklahoma State University from January 2003 to August 2003. Teaching Assistant in the Department of Computer Science, Oklahoma State University from August 2003 to May 2004. Internship in Seagate Technology, Oklahoma city summer 2003.

Professional Memberships: Association for Computing Machinery.

Name: Sudheer Krishna Chimbli Venkata Date of Degree: December, 2004.

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: Sector Based Clustering & Routing

Pages in Study: 194 Candidate for the Degree of Master of Science

Major Field: Computer Science

Scope and Method of Study: The purpose of this research is to provide a new secure and efficient protocol for clustering and routing in Mobile Ad hoc Networks. The existing protocols for MANET have poor performance when the power and security attributes are considered. A new protocol that would take these two attributes along with performance into consideration is required. In our approach the concept of sectors is introduced to the clustering and routing process. The concept of sectors facilitates in the cluster head election process and also sets an upper limit on the number of members a cluster can have. In Sector Based Routing the route discovery mechanism chooses routes in a controlled way using the concept of sectors. The multiple paths help in reducing the power consumption and increase the security of data being transmitted.

Findings and Conclusion: Simulations were performed on the OSU Computer Science Department Sun Blade 150 server. The simulator was specially designed and programmed to simulate and compare the proposed Sector Based Clustering and Routing (SBCR) algorithms along with Ad hoc On-Demand Distance Vector Routing (AODV) and Cluster Based Routing (CBR) protocols. Two types of simulations were performed on the three protocols. One simulation tested the survivability of the network whereas the other observed the behavior of the protocol over a long period of time. The proposed protocol increases the survivability of the network, improves the performance, as well as security. The results suggest that as the node density increases, our proposed sector based clustering and routing outperforms AODV and CBR. Additional simulation results were also collected to observe the behavior of SBCR, with an increase in the number of sectors. As the number of sectors was increased the performance of the network improved.

Advisor's Approval: ______________________________